UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Aprendizaje Semi-supervisado con Modelos Generativos Profundos

Autor: Íñigo Álvaro Sáenz
Tutor: Daniel Hernández Lobato

junio 2019

Íñigo Álvaro Sáenz
Aprendizaje Semi-supervisado con Modelos Generativos Profundos

Íñigo Álvaro Sáenz

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Con este trabajo de fin de grado pretendemos mostrar como implementar modelos generativos profundos utilizando técnicas de inferencia aproximada para realizar aprendizaje semi supervisado.

Al tratarse de aprendizaje semi supervisado, solo contamos con un pequeño porcentaje de las etiquetas de los datos de entrenamiento. De esta forma, este tipo de modelos se convierten en una opción muy interesante en la práctica, al reducir en tiempo y recursos el proceso de etiquetado inicial de los datos de entrenamiento, ya que tan solo con un pequeño subconjunto de las etiquetas son capaces de obtener buenos resultados. Nuestro estudio se va a limitar a su aplicación sobre el conjunto de datos de dígitos escritos MNIST, es decir, para realizar clasificación de imágenes.

Nuestra red hará uso de modelos probabilísticos para extraer una serie de atributos de la imagen original. De este modo, vamos a crear un codificador ('encoder') que a partir del dato original, y sin necesidad de conocer la clase, "codificará" la imagen original en una representación de alto nivel y menor dimensionalidad. A su vez, esta representación puede decodificarse a través de un 'decoder' para generar nuevas imágenes parecidas a la original. Además, en el proceso de aprendizaje de esos atributos ocultos, aprenderemos al mismo tiempo los parámetros del clasificador que llevará a cabo las tareas de discriminación entre las clases.

Estos modelos obtienen resultados muy positivos en comparación a otras técnicas del estado del arte. Esta diferencia se torna más abrupta cuanto menor es el porcentaje de datos etiquetados, poniendo de manifiesto que la técnica planteada en este trabajo es una de las mejores opciones a la hora de abordar un problema de aprendizaje semi supervisado.

Para el desarrollo de los modelos y las pruebas hemos utilizado Python 3 y el framework Tensorflow. Para comprobar su funcionamiento y rendimiento hemos realizado los experimentos sobre el conjunto de datos MNIST, en los que hemos ido variando el número de datos etiquetados. Todos los modelos basados en redes neuronales descritos en esta memoria han sido implementados desde cero usando los lenguajes y frameworks descritos.

Palabras clave

aprendizaje semi supervisado, aprendizaje profundo, Tensorflow, MNIST, variables latentes

Abstract

The main goal is to show how to implement deep generative models applying approximate variational inference techniques in order to perform semi-supervised learning.

In semi-supervised learning, we only use a small percentage of the training data labels. This way, these models become a really interesting option, as they reduce in time and resources the initial labelling process since they only need a small subset of class labels to get good performance results. We are going to limit this study to the images data set of hand written digits MNIST, that is, for doing image classification.

The networks will use probabilistic models to infer a series of attributes from the original image. Thus, we are going to create an 'encoder' that, from the original data, and without needing to know the label, will 'encode' the original image into a high level representation with lower dimension. We can 'decode' that representation later through a 'decoder' in order to generate new images similar to the original ones. Furthermore, in the learning process of these codifications, we will also be able to learn the parameters of a discriminative classifier, used to make future predictions.

These models perform really well compared to other state of the art techniques. In fact, the biggest gap was found when the number of available labels was limited to a small percentage. This way, our models become one of the best options when we are facing a semi supervised learnig problem.

We've used Python 3 and the framework TensorFlow to develop and test the models. The experiments were carried out with the MNIST data set, varying the number of labeled data. All the models based on neuronal networks described in this document were fully implemented from the beginning using the languages and frameworks described.

Keywords

semi supervised learning, deep learning, Tensorflow, MNIST, latent variables

Índice

1 Introducción **1**

1.1 Motivación 1

1.2 Objetivos 2

1.3 Organización de la memoria 3

2 Estado del arte **5**

2.1 Trabajo relacionado en aprendizaje semi supervisado 5

3 Marco teórico **7**

3.1 Variational auto-encoders 7

3.2 Modelos generativos profundos para aprendizaje semi supervisado 8

3.3 Inferencia Variacional 11

3.4 Optimización con Lower Bound 12

3.5 Truco de reparametrización 15

4 Implementación **17**

4.1 Conjunto de datos MNIST 17

4.2 Implementación M1 18

4.3 Implementación M2 24

4.4 Implementación M1 + M2 32

5 Pruebas y resultados **33**

5.1 Comparativa de los resultados 33

5.2 Resultados M1 34

5.3 Resultados M2 36

5.4 Resultados M1+M2 36

6 Conclusiones **39**

6.1 Conclusiones y trabajo futuro 39

Bibliografía **42**

Listas

Lista de códigos

4.1 Red neuronal encoder M1 . . . 19
4.2 Método para aplicar el truco de reparametrización . . . 20
4.3 Cálculo del $\mathcal{L}$ por mini batch en el modelo M1 . . . 22
4.4 Optimización del $\mathcal{L}$ en el modelo M1 . . . 23
4.5 Bucle de entrenamiento del modelo M1 . . . 24
4.6 Redes neuronales del encoder M2 . . . 26
4.7 Método para calcular $\mathcal{L}$ de los datos etiquetados . . . 29
4.8 Método para calcular $\mathcal{U}$ de los datos no etiquetados . . . 30

Lista de ecuaciones

3.1 Modelo generativo VAE . . . 7
3.2 Encoder VAE . . . 8
3.3 Modelo generativo M1 . . . 9
3.4 Encoder M1 . . . 9
3.5 Modelo generativo M2 . . . 10
3.6 Clasificador modelo M2 . . . 10
3.7 Encoder M2 . . . 10
3.8 Descomposición de la verosimilitud marginal en inferencia variacional . . . 11
3.9 Relación Lower Bound con la divergencia de Kullback-Leibler y verosimilitud marginal 11
3.10 Descomposición de la verosimilitud marginal en el modelo M1 . . . 12
3.11 Lower Bound utilizando esperanzas . . . 13
3.12 Lower Bound para el modelo M1 . . . 13
3.16 Lower Bound con minibatches . . . 14
3.17 Lower Bound para el modelo M2 con datos etiquetados . . . 14
3.18 Lower Bound para el modelo M2 con datos sin etiquetar . . . 14
3.20 Función objetivo del modelo M2 . . . 15

Lista de figuras

1.1 Interés en deep learning . . . 1

3.1 VAE gráfico 8
3.2 Concepto Lower Bound 12

5.1 Resultados cualitativos modelo M1 35
5.2 Evolución del Lower Bound en los experimentos del modelo M1 35
5.3 Resultados del entrenamiento del M2 37
5.4 Resultados del entrenamiento del M1+M2 38

1

Introducción

1.1. Motivación

A lo largo de los últimos años hemos sido testigos del incremento del interés en Deep Learning debido a sus múltiples aplicaciones, como visión por computador [1] o procesamiento de lenguaje natural [2], y su excelente rendimiento, superior al de otras aproximaciones del estado del arte. Podemos ver reflejado esto en la gráfica 1.1, extraída de Google Trends y que muestra la popularidad del término de búsqueda "deep learning".

Figura 1.1: Índice de popularidad del término de búsqueda 'deep learning' en Google a lo largo del tiempo.

Además, la continua mejora del hardware, en concreto de GPU, también ha propiciado este aumento de interés. Ahora somos capaces de reducir los tiempos de cálculo de esos algoritmos sin tener que recurrir a un supercomputador para poder implementarlos. De esta forma, estas técnicas se han vuelto más accesible para un mayor número de personas.

Este auge ha tenido gran repercusión en el mundo empresarial. Las empresas se han dado cuenta del enorme valor que les aporta y es por eso que los data scientist se han vuelto una de las profesiones más solicitadas [3].

Sin embargo, para llevar a cabo tareas de clasificación y obtener buenos resultados, en muchas ocasiones es importante contar con un conjunto de entrenamiento etiquetado de un tamaño conside-

rable. El acceso a los datos de ejemplo no resulta problemático, vivimos en la época del Big Data y tenemos almacenadas enormes cantidades de datos. El problema aparece a la hora de encontrar las etiquetas de las clases, pues esta tarea resulta costosa tanto en tiempo como en recursos al ser necesaria la intervención de una persona para realizarlo. De este modo, el aprendizaje semi supervisado aparece como una alternativa muy interesante, pues utilizando un pequeño porcentaje de los datos de entrenamiento etiquetados y ayudado por el resto sin etiquetar [4], son capaces de obtener muy buenos resultados.

Además esta aproximación está muy relacionada con la forma que tenemos las personas de aprender, ya que cuando aprendemos un concepto, no necesitamos que nos indiquen cada vez que vemos un nuevo ejemplo de que se trata, sino que somos capaces de extraer características comunes y relacionar los ejemplos entre sí para saber que se trata de lo mismo. Philip Haeusser, Alexander Mordvintsev y Daniel Cremers [5] explican esta misma idea:

> " *Philip Haeusser, Alexander Mordvintsev and Daniel Cremers: A child is able to learn new concepts quickly and without the need for millions examples that are pointed out individually. Once a child has seen one dog, she or he will be able to recognize other dogs and becomes better at recognition with subsequent exposure to more variety.* "

1.2. Objetivos

Aunque el aprendizaje semi supervisado se puede utilizar tanto para clustering como para clasificación, nosotros vamos a centrarnos en esto último. Por lo tanto, el objetivo principal de este TFG consistirá en implementar con Tensorflow un modelo de aprendizaje semi supervisado para realizar clasificación de imágenes. En concreto, nos hemos decantado por utilizar un modelo generativo profundo basado en los avances en inferencia variacional realizados por Diederik P. Kingma en su artículo *Semi Supervised Learning with Deep Generative Models* [6]. En ellos, el autor propone un método que hará uso de las variables latentes del conjunto de datos. El problema radica en que las probabilidades posteriores de estas variables son intratables, por lo que es necesario recurrir a una aproximación. En su artículo, el autor define una estimación del *lower bound* que nos permitirá realizar inferencia aproximada sobre la probabilidad posterior de las variables latentes del conjunto de datos al mismo tiempo que aprendemos los parámetros del modelo.

Además de la implementación del modelo, compararemos los resultados obtenidos por nuestro modelo con el de otros modelos de aprendizaje semi supervisado ya publicados.

1.3. Organización de la memoria

Comenzaremos la memoria comentando el estado del arte del aprendizaje semi supervisado en el capítulo 2, enumerando otras técnicas utilizadas y algunos de sus usos prácticos. Continuaremos en el capítulo 3 definiendo los modelos que vamos a implementar y exponiendo el marco teórico planteado por Diederik P. Kingma en [6] que sustenta su funcionamiento, para después, en el capítulo 4, explicar una posible implementación en TensorFlow de los mismos, cuyos resultados mostraremos en el capítulo 5, en el que además compararemos con las técnicas mencionadas en el estado del arte. Por último, en el capítulo 6, exponemos nuestras conclusiones y trabajo futuro respecto a los modelos.

2

Estado del arte

2.1. Trabajo relacionado en aprendizaje semi supervisado

Existen numerosas aplicaciones en las que resulta muy conveniente utilizar esta técnica. Todas ellas tienen en común el hecho de que a pesar de que el volumen de datos de ejemplo que explotar es muy amplio, la obtención de etiquetas de la clase es un proceso costoso. Ejemplos de estos casos pueden ser clasificación de páginas web [7], procesamiento de lenguaje natural [8] o incluso biomedicina [9]. Debido a esto, el aprendizaje semi supervisado ha sido objeto de muchos estudios y se han desarrollado diversas técnicas para llevarlo a cabo. Entre ellas sobresalen:

Técnicas de auto etiquetado (*Self-Labelling techniques*) [10]: este enfoque trata de ampliar el número de datos etiquetados añadiendo sus propias predicciones. El problema principal de estos modelos es que si las predicciones tienden a ser erróneas, el modelo estará basando su conocimiento en datos incorrectos. Además, son difícilmente escalables ya que cuantos mas datos de entrenamiento haya, mas probabilidad de que alguna de las clases auto etiquetadas por el modelo sea errónea.

TSVM (*Transductive* SVM): son una extensión de las SVM, es por ello que conviene hablar primero de las *Support Vector Machines*. Se trata de un modelo de aprendizaje supervisado cuyo objetivo se basa en separar los espacios a los que pertenecen las distintas clases mediante hiperplanos [11]. Las TSVM [12] aplican este mismo concepto pero utilizando además los datos sin etiquetar.

CAE (*Contrastive Auto Encoders*): su funcionamiento consiste en crear varios *auto encoders* independientes y después utilizar sus diferencias para construir su función objetivo [13].

CNN : Esta es una de las aproximaciones que más ha evolucionado en los últimos años. Aunque existen diversos algoritmos para implementar aprendizaje semi supervisado utilizando redes convolucionales, uno de los más recientes y prometedores son las redes neuronales que aprenden por 'asociación' [5]. En el artículo citado, los autores proponen un método basado en la generación de *embeddings* de los datos tanto etiquetados como no etiquetados, suponiendo que la red genera *embeddings* parecidos en los datos de la misma clase.

Después, un '*walker*' trata de recorrer el dataset en zig zag yendo desde los *embeddings* generados a partir de los datos etiquetados a otro del no etiquetado, premiando terminar en la misma clase en la que empezó.

Modelos Generativos : basan su funcionamiento en modelos probabilísticos, a partir de los cuales podemos muestrear nuevos puntos similares a los originales. Además, estos modelos son capaces de generar nuevas representaciones de los datos, que nos ayudan en la tarea de clasificación.

Nosotros hemos optado por poner en práctica técnicas de aprendizaje semi supervisado basadas en modelos generativos, ya que siguen siendo una de las opciones que mejores resultados arrojan. A lo largo de las siguientes secciones explicaremos a fondo su funcionamiento y posible implementación.

3

Marco teórico

3.1. Variational auto-encoders

Antes de entrar en profundidad a explicar el marco teórico que fundamenta nuestros modelos, merece la pena mencionar el concepto de *variational auto-encoder* (VAE), ya que están íntimamente relacionados.

Un VAE es un modelo generativo de aprendizaje no supervisado basado en el uso de variables latentes. Su objetivo principal consiste en estimar la distribución de los puntos de ejemplo $p_\theta(\boldsymbol{x})$ para alcanzar un mayor nivel de comprensión sobre los mismos. De este modo, somos capaces de generar nuevos puntos similares a los originales (lo que tiene además otros beneficios prácticos) a través de ella. La técnica ideal para aprender $p_\theta(\boldsymbol{x})$ sería por máxima verosimilitud, maximizando respecto de $\boldsymbol{\theta}$ la probabilidad de los datos observados. Sin embargo, esta probabilidad es intratable. En este punto entran en juego las variables latentes, que nos permiten escribir $p_\theta(\boldsymbol{x})$ como:

$$p_\theta(\boldsymbol{x}) = \int p_\theta(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})dz$$

Básicamente, la ecuación de más arriba define la generación de los puntos $\boldsymbol{X}$ como un proceso que depende de las variables latentes, de modo que, si somos capaces de muestrearlas, podemos llegar a generar nuevos puntos $\boldsymbol{x}$ parecidos a los originales a través de $p_\theta(\boldsymbol{x}|\boldsymbol{z})$. Estas variables latentes representan características de alto nivel del dato original con una dimensionalidad menor. Cabe destacar que estas variables no se observan, sino que se infieren de los datos observables.

Para generar las variables latentes los VAE utilizan la distribución $p(\boldsymbol{z}|\boldsymbol{x})$ que maximice la probabilidad de las $\boldsymbol{z}$ dados los puntos observados. Esta distribución recibe el nombre de *encoder* del modelo. Para implementarlo recurrimos a la aproximación de la verdadera probabilidad $q_\phi(\boldsymbol{z}|\boldsymbol{x})$. Una vez que hacemos el muestreo de variables latentes a través de la distribución 3.2, utilizamos el *decoder* para generar puntos similares a los $\boldsymbol{X}$ originales. Este *decoder* se corresponde con la distribución $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, a partir de la cual generamos los nuevos puntos. De este modo, el VAE queda formado por el *encoder* y el *decoder* que vemos en las ecuaciones 3.2 y 3.1 respectivamente.

$$p_\theta(\boldsymbol{x}|\boldsymbol{z}) \tag{3.1}$$

$$q_\phi(\boldsymbol{z}|\boldsymbol{x}) \approx p(\boldsymbol{z}|\boldsymbol{x}) \tag{3.2}$$

Al introducir las probabilidades 3.2 y 3.1 podemos inferir los parámetros de las distribuciones del VAE utilizando inferencia variacional y así solventar el problema de la intratabilidad. Describiremos este método en detalle en la sección 3.3.

Por otra parte, cabe destacar que los parámetros de las distribuciones son el resultado de redes neuronales. Debemos hacer hincapié en que los parámetros del *encoder* y el *decoder*, $\boldsymbol{\theta}$ y $\boldsymbol{\phi}$, difieren, puesto que son relativos a distintas distribuciones. Podemos ver este proceso de forma más intuitiva y gráfica en la imagen 3.1, extraída del artículo *Auto-encoding variational bayes* [14].

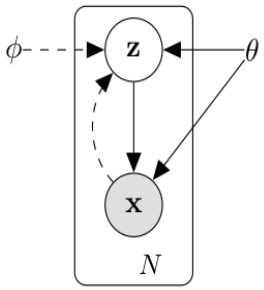


Figura 3.1: Las líneas continuas representan el modelo generativo 3.1 y las discontinuas el *encoder* 3.2, es decir, la aproximación al verdadero posterior $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ [14]. Cabe destacar que los parámetros θ y ϕ se aprenden al mismo tiempo.

3.2. Modelos generativos profundos para aprendizaje semi supervisado

En esta sección vamos a enunciar los modelos planteados por Diederik P. Kingma en su artículo *Semi Supervised Learning with Deep Generative Models* [6] para después explicar con mayor detenimiento su fundamento. Todos ellos tienen en común el hecho de que explotan las variables latentes de los datos y que para ello parten de un modelo generativo y un *encoder* (al que también nos referiremos como modelo de reconocimiento), como en el caso de los VAE 3.1. Del mismo modo, la verdadera probabilidad posterior no se puede calcular, por lo que debemos recurrir a una aproximación. Sin embargo, el VAE tiene numerosas limitaciones que estos modelos solventan [14]:

- El tamaño del conjunto de datos.
- Intratabilidad. Tanto de la *marginal likelihood* $p_\theta(\boldsymbol{x}) = \int p_\theta(\boldsymbol{x}|\boldsymbol{z})p_\theta(\boldsymbol{z})dz$ como de la verdadera probabilidad posterior $p_\theta(\boldsymbol{z}|\boldsymbol{x}) = p_\theta(\boldsymbol{x}|\boldsymbol{z})p_\theta(\boldsymbol{z})/p_\theta(\boldsymbol{x})$ o de cualquier integral que fuera necesaria.

Además, este enfoque es especialmente potente por el hecho de aprender al mismo tiempo los parámetros de la aproximación del posterior 3.2 $\boldsymbol{\phi}$, como del modelo generativo 3.1 $\boldsymbol{\theta}$ y las variables latentes $\boldsymbol{Z}$.

Veremos el método diseñado por el autor para hacer esto en las siguientes secciones, ahora vamos a centrarnos en comprender los objetivos y bases de cada uno de los modelos utilizados:

Modelo discriminatorio de características latentes (*'Latent feature discriminative model'*):
A lo largo de la memoria nos referiremos a este modelo como **M1** siguiendo la notación del autor. Cabe recalcar que este modelo no se enmarca dentro del aprendizaje semi supervisado, sino no supervisado, ya que no hace uso de las etiquetas. Además, no es un clasificador *per se*, ya que su cometido es únicamente el de obtener las variables latentes de los puntos originales para posteriormente utilizarlos como input de otro modelo capaz de clasificar. Del mismo modo, podemos utilizar el *decoder* para generar nuevos puntos similares a los originales. El modelo generativo es 3.3:

$$p_\theta(\boldsymbol{x}|\boldsymbol{z}) = f(\boldsymbol{x};\boldsymbol{z},\boldsymbol{\theta}) \qquad p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{0},\boldsymbol{I}) \tag{3.3}$$

Donde $f(\boldsymbol{x};\boldsymbol{z},\boldsymbol{\theta})$ puede ser cualquier función de verosimilitud en la que sus probabilidades están formadas por una serie de transformaciones, con parámetros $\boldsymbol{\theta}$, sobre las variables latentes inferidas. Para realizar estas transformaciones utilizamos redes neuronales que, con las variables latentes como entrada, producen los parámetros de la distribución $p_\theta(\boldsymbol{x}|\boldsymbol{z})$. Por ejemplo, si fuera una distribución de Bernoulli tendríamos un MLP con función de activacíon *softmax* en la salida. En la sección 4 detallaremos por qué finalmente escogimos una distribución de Bernoulli para este caso.
Respecto al modelo de reconocimiento $p(\boldsymbol{z}|\boldsymbol{x})$, volvemos a hacer hincapié en la imposibilidad de calcular su valor real, por lo que tendremos que recurrir a la aproximación 3.4.

$$q_\phi(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_\phi(\boldsymbol{x}), diag(\boldsymbol{\sigma}^2_\phi(\boldsymbol{x}))) \tag{3.4}$$

De nuevo, utilizaremos una red neuronal para, con los puntos originales como entrada, generar los parámetros de la distribución de las variables latentes. Esto es precisamente lo que indica la notación $\boldsymbol{\mu}_\phi(\boldsymbol{x})$ y $\boldsymbol{\sigma}^2_\phi(\boldsymbol{x})$, en el primer caso, que la media está generada por un MLP con las $\boldsymbol{x}$ como entrada y en el segundo, que la varianza esta generada por un

MLP con las $\boldsymbol{x}$ como entrada.

Modelo generativo semi supervisado (*'Generative semi-supervised model'*) o **M2**: El principio fundamental de este modelo es similar al del M1, pero en este caso si que hacemos uso de una parte de las etiquetas para el entrenamiento. Asimismo, además de aprender a generar las variables latentes de los datos, somos capaces de aprender los parámetros del clasificador, lo que nos permitirá realizar predicciones sobre nuevos puntos sin necesidad de recurrir a otro modelo (como en el caso del M1), que haciendo uso de las variables latentes, realice esta tarea. La idea subyacente de este modelo y la que lo diferencia del M1, radica en que en este caso los puntos del conjunto de datos están relacionados con las variables latentes continuas $\boldsymbol{z}$ y además, con variables latentes de clase y, como vemos en las ecuaciones 3.5:

$$p_\theta(\boldsymbol{x}|y,\boldsymbol{z}) = f(\boldsymbol{x};y,\boldsymbol{z},\boldsymbol{\theta}) \qquad p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{0},\boldsymbol{I}) \qquad p(y) = Cat(y|\boldsymbol{\pi}) \tag{3.5}$$

Donde $Cat(y|\boldsymbol{\pi})$ es la distribución Multinomial con parámetro $\boldsymbol{\pi}$, $f(\boldsymbol{x};y,\boldsymbol{z},\boldsymbol{\theta})$ es una función de verosimilitud válida (cuyos parámetros son la salida de un MLP) y el prior de las $\boldsymbol{z}$ es una Gaussiana estándar. Lo interesante de este modelo, es que cuando no conocemos las etiquetas del punto de entrenamiento, es decir, cuando utilizamos el subconjunto no etiquetado, tratamos a las clases como variables latentes, por lo que también estamos realizando inferencia sobre ellas además de sobre las variables latentes. De esta forma, estamos clasificando con inferencia. Las predicciones se obtienen a través $p(y|\boldsymbol{x})$. Sin embargo, como en el caso de la distribución de las variables latentes, no podemos calcular el posterior real, por lo que tenemos que recurrir a la aproximación 3.6, cuyo parámetro $\boldsymbol{\pi}$ es el resultado de una red neuronal que tiene como entrada los datos originales $\boldsymbol{x}$.

$$q_\phi(y|\boldsymbol{x}) = Cat(y|\boldsymbol{\pi}_\phi(\boldsymbol{x})) \tag{3.6}$$

Por último, nos queda hablar del modelo de reconocimiento $p(\boldsymbol{z}|\boldsymbol{x})$, para el que nuevamente hay que recurrir a una aproximación, como vemos en 3.7. Si observamos con detenimiento la ecuación, podremos percatarnos de que en este caso la media de la distribución depende tanto de $\boldsymbol{x}$ como de $\boldsymbol{y}$, mientras que la varianza depende solo de $\boldsymbol{x}$. Tendremos que tener esto en cuenta a la hora de crear los MLP que generan los parámetros de la distribución. De esta forma, la media será el resultado de una red que tenga como entrada $\boldsymbol{x}$ e $\boldsymbol{y}$, y la varianza, el de otra red con solo las $\boldsymbol{x}$ como *input*.

$$q_\phi(\boldsymbol{z}|y,\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_\phi(y,\boldsymbol{x}), diag(\boldsymbol{\sigma}^2_\phi(\boldsymbol{x}))) \tag{3.7}$$

Cabe reseñar que hacemos esta descomposición porque asumimos que el modelo de re-

conocimiento tiene la forma $q_\phi(\boldsymbol{z}, y|\boldsymbol{x}) = q_\phi(\boldsymbol{z}|\boldsymbol{x})q_\phi(y|\boldsymbol{x})$.

Modelo generativo semi supervisado apilado (**M1 + M2**): Este modelo combina los dos modelos M1 y M2. La idea consiste en utilizar las variables latentes inferidas por el modelo M1 (z_1) para clasificar utilizando el M2. El resultado es un modelo generativo de dos capas: $p_\theta(\boldsymbol{x}, y, \boldsymbol{z_1}, \boldsymbol{z_2}) = p(y)p(\boldsymbol{z_2})p_\theta(\boldsymbol{x}|\boldsymbol{z_1})p_\theta(\boldsymbol{z_1}|y, \boldsymbol{z_2})$, donde $p(y)$ y $p(\boldsymbol{z_2})$ son iguales que en el modelo M2, $p_\theta(\boldsymbol{x}|\boldsymbol{z_1})$ es el modelo generativo del M1 3.3 y $p_\theta(\boldsymbol{z_1}|y, \boldsymbol{z_2})$ es el del M2, con la particularidad de que en este caso la entrada son las variables latentes del primer modelo $\boldsymbol{z_1}$ y entendiendo como $\boldsymbol{z_2}$ las del M2. De este modo, la distribución del modelo generativo de la segunda capa $p_\theta(\boldsymbol{z_1}|, \boldsymbol{z_2})$ ya no sigue una distribución de Bernoulli como en el M2, sino que será una Gaussiana, pues los datos son reales (variables latentes) y no las probabilidades de activación de cada píxel. Los *encoder* son exactamente iguales que en los modelos anteriores 3.4 para generar $\boldsymbol{z_1}$ y 3.7 para obtener $\boldsymbol{z_2}$. Por último, el modelo de predicción es equivalente al caso del M2 3.6 pero con $\boldsymbol{z_1}$ en lugar de las $\boldsymbol{x}$ originales, que solo se utilizan para obtener la primera capa de variables latentes. También merece la pena destacar que los modelos se ejecutan de forma secuencial, es decir, primero generamos las variables latentes utilizando el M1, para posteriormente introducirlas en el modelo M2. En la práctica, no es necesario utilizar la primera capa de variables latentes en sí, pues en algunos casos es suficiente con la media de su distribución.

En todos los casos hemos visto como es necesario recurrir a una aproximación de la verdadera probabilidad posterior para los modelos de reconocimiento. Para aproximar la distribución utilizaremos inferencia variacional. En la siguiente sección detallaremos el funcionamiento de este método.

3.3. Inferencia Variacional

Para estimar el valor de los parámetros $\boldsymbol{\theta}$ y $\boldsymbol{\phi}$, utilizamos inferencia variacional [15]. Con el fin de explicar sus principios, comenzaremos descomponiendo la verosimilitud marginal de un problema similar en el que queremos calcular $\log p_\theta(\boldsymbol{x})$:

$$\log p_\theta(\boldsymbol{x}) = KL(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})) + \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; x) \tag{3.8}$$

Donde KL es la divergencia de Kullback-Leibler, que mide la similitud entre dos distribuciones. Su valor es siempre positivo menos cuando $p = q$, que toma el valor 0. De esta forma, el término $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; x)$ actúa como Lower Bound. Podemos ver esto mejor reescribiendo la ecuación como en 3.9 y de forma gráfica en la figura 3.2.

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; x) = \log p_\theta(\boldsymbol{x}) - KL(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x})) \tag{3.9}$$

Como KL será cero cuando ambas distribuciones sean iguales, según nos acerquemos a ese valor $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; x)$ será cada vez más similar a $p_\theta(\boldsymbol{x})$, luego maximizar el Lower Bound respecto de $\boldsymbol{\theta}$, tiene como resultado maximizar también $p_\theta(\boldsymbol{x})$. De esta forma, conseguimos una función objetivo tanto para $\boldsymbol{\theta}$ como para $\boldsymbol{\phi}$. En consecuencia, en el proceso de aprendizaje del modelo generativo somos capaces de aprender al mismo tiempo el de reconocimiento $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ sin ningún esfuerzo extra.

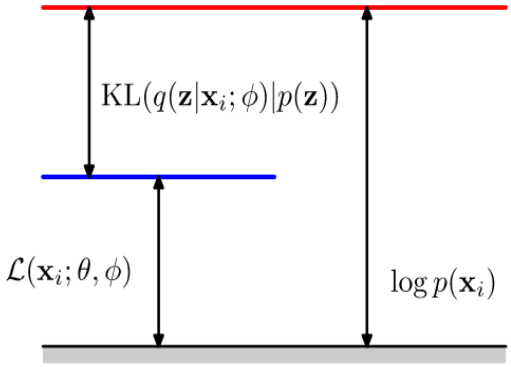


Figura 3.2: La figura, extraída de *Importance Weighted Autoencoders with Uncertain Neural Network Weights* [16], nos muestra la relación entre el logaritmo de la verosimilitud marginal $\log p_\theta(\boldsymbol{x})$ respecto los términos $KL(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}|\boldsymbol{x}))$ y $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; x)$. Cuanto mayor sea la similitud entre p y q, menor será $KL(q||p)$ y mayor será el Lower Bound, pero siempre dentro de los límites marcados por $\log p(\boldsymbol{x})$). De esta forma, maximizar el Lower Bound supone maximizar también $\log p_\theta(\boldsymbol{x})$, infiriendo tanto $\boldsymbol{\theta}$ como $\boldsymbol{\phi}$ en el proceso.

3.4. Optimización con Lower Bound

Ahora vamos a aplicar las fórmulas de la sección 3.3 en nuestros modelos para derivar una función objetivo que se pueda calcular y que nos permita aprender al mismo tiempo los parámetros de los modelos de reconocimiento (*encoder*) y de los generativos, ϕ y θ respectivamente. El método está basado en *stochastic gradient variational Bayes* [14] y las ecuaciones para obtener la función objetivo del modelo M1 se encuentran en ese artículo. La función objetivo del modelo M2 se obtuvo de *Semi-Supervised Learning with Deep Generative Models* [6].

3.4.1. Función objetivo del modelo discriminatorio de características latentes

Si introducimos las probabilidades de los modelos generativos y de reconocimiento del M1 dentro de la ecuación 3.8 obtenemos 3.10, es decir, la descomposición de la verosimilitud marginal por cada punto del conjunto de datos i:

$$\log p_\theta(\boldsymbol{x}^{(i)}) = KL(q_\phi(\boldsymbol{z}|\boldsymbol{x}^{(i)})||p_\theta(\boldsymbol{z}|\boldsymbol{x}^{(i)})) + \mathcal{L}(\theta, \phi; \boldsymbol{x}^{(i)}) \tag{3.10}$$

El Lower Bound se puede escribir utilizando esperanzas como en 3.11. De esta forma, podemos calcular su valor con algún método de aproximación.

$$\log p_\theta(\boldsymbol{x}^{(i)}) \geq \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; x^{(i)}) = \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[-\log q_\phi(\boldsymbol{z}|\boldsymbol{x}) + \log p_\theta(\boldsymbol{x}, \boldsymbol{z})] \tag{3.11}$$

3.11 se puede reescribir como:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}^{(i)}) = -KL(q_\phi(\boldsymbol{z}|\boldsymbol{x}^{(i)})||p_\theta(\boldsymbol{z})) + \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x}^{(i)})}[\log p_\theta(\boldsymbol{x}^{(i)}|\boldsymbol{z})] \tag{3.12}$$

Esta es la función objetivo de nuestro modelo, sobre la que debemos optimizar y diferenciar respecto a los parámetros de los modelos de reconocimiento y generativos, ϕ y $\boldsymbol{\theta}$ respectivamente. El autor define un método para estimar su valor, al que denomina *stochastic gradient variational Bayes* (**SGVB**) y un algoritmo para llevarlo a cabo: *Auto Encoding variational Bayes* (**AEVB**) [14]. Este método se centra en encontrar una forma de estimar el valor de $\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|\boldsymbol{z})]$ en 3.12, ya que el cálculo de la divergencia KL no es tan problemático, como veremos a lo largo de la sección. Para ello, utilizaremos aproximación mediante re-muestreo. Esto requiere que seamos capaces de calcular $p_\theta(\boldsymbol{x}|\boldsymbol{z})$. En este contexto, el autor propone un método para generar muestras de los valores de las variables latentes a partir de $q_\phi(\boldsymbol{z}|\boldsymbol{x})$, de modo que $\tilde{\boldsymbol{z}} \sim q_\phi(\boldsymbol{\epsilon}, \boldsymbol{x})$, a partir de una transformación diferenciable $g_\phi(\boldsymbol{\epsilon}, \boldsymbol{x})$ de una variable de ruido auxiliar $\boldsymbol{\epsilon}$. Kingma se refiere a esto como *reparametrization trick* [14] y dedicaremos la sección 3.5 a su explicación. De este modo, ahora podemos formar estimaciones por Monte Carlo de la esperanza para obtener 3.13.

$$\tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}^{(i)}) = -KL(q_\phi(\boldsymbol{z}|\boldsymbol{x}^{(i)})||p_\theta(\boldsymbol{z})) + \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(\boldsymbol{x}^{(i)}|\boldsymbol{z}^{(i,l)}) \tag{3.13}$$

Donde L son las muestras generadas de variables latentes por cada punto $\boldsymbol{x}$ del conjunto de datos.

Por otra parte, para calcular la divergencia de KL, hemos utilizado la solución propuesta en [14], que podemos ver en 3.14:

$$-KL(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z})) = \frac{1}{2}\sum_{j=1}^{J}(1 + \log \sigma_j^2(x) - \boldsymbol{\mu}_j^2(\boldsymbol{x}) - \boldsymbol{\sigma}_j^2(\boldsymbol{x})) \tag{3.14}$$

Donde J es la dimensionalidad de las variables latentes generadas por punto. De esta forma, el

Lower Bound para cada punto i queda de la siguiente manera:

$$\tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}^{(i)}) = \frac{1}{2}\sum_{j=1}^{J}(1 + \log \boldsymbol{\sigma}_j^2(\boldsymbol{x}^{(i)}) - \boldsymbol{\mu}_j^2(\boldsymbol{x}^{(i)}) - \boldsymbol{\sigma}_j^2(\boldsymbol{x}^{(i)})) + \frac{1}{L}\sum_{l=1}^{L}\log p_\theta(\boldsymbol{x}^{(i)}|\boldsymbol{z}^{(i,l)}) \tag{3.15}$$

Como el tamaño del conjunto de datos que vamos a utilizar es relativamente grande (60000 puntos), merece la pena utilizar mini batches para calcular el Lower Bound. Para ello utilizaremos 3.16, en donde M es el tamaño del mini batch y N el del número total de puntos. De este modo, el sumatorio recorre los puntos del mini batch.

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{X}) \simeq \tilde{\mathcal{L}}^M(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{X}^M) = \frac{N}{M}\sum_{i=1}^{M}\tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}^{(i)}) \tag{3.16}$$

3.4.2. Función objetivo del modelo generativo semi supervisado

En el caso de este modelo, para derivar el Lower Bound, tenemos que hacer distinción entre cuando los datos tienen etiquetas y cuando no las tienen. Cuando tenemos las etiquetas, la función objetivo es una extensión de 3.12:

$$-\mathcal{L}(\boldsymbol{x}, y) = \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x},y)}[\log p_\theta(\boldsymbol{x}|y,\boldsymbol{z}) + \log p_\theta(y) + \log p(\boldsymbol{z}) - \log q_\phi(\boldsymbol{z}|\boldsymbol{x},y)] \tag{3.17}$$

Cuando no tenemos las etiquetas, tratamos a la clase como otra variable latente sobre la que realizar inferencia.

$$\begin{aligned}\log p_\theta(\boldsymbol{x}) &\geq \mathbb{E}_{q_\phi(\boldsymbol{z},y|\boldsymbol{x})}[\log p_\theta(\boldsymbol{x}|y,\boldsymbol{z}) + \log p_\theta(y) + \log p_\theta(\boldsymbol{z}) - \log q_\phi(\boldsymbol{z},y|\boldsymbol{x})] \\ &= \sum_y q_\phi(y|\boldsymbol{x})(-\mathcal{L}(\boldsymbol{x},y)) + \mathcal{H}(q_\phi(y|\boldsymbol{x})) = -\mathcal{U}(\boldsymbol{x})\end{aligned} \tag{3.18}$$

siendo $\mathcal{H}$ la entropía. El Lower Bound para el conjunto de datos completo se corresponde con:

$$\mathcal{J} = \sum_{(\boldsymbol{x},y)\sim\tilde{p}_l}\mathcal{L}(\boldsymbol{x},y) + \sum_{(\boldsymbol{x})\sim\tilde{p_u}}\mathcal{U}(\boldsymbol{x}) \tag{3.19}$$

Por último, para mejorar la capacidad de aprendizaje del clasificador ($q_\phi(y|\boldsymbol{x})$), el autor propone añadir una pérdida de clasificación sobre los datos etiquetados a la función objetivo 3.19, ya que $q_\phi(y|x)$ aparece únicamente en el segundo término (el relativo a los datos sin etiquetar, $\mathcal{U}(\boldsymbol{x})$). De esta forma, nuestro clasificador aprenderá además de los datos etiquetados y quedará de la siguiente

forma:

$$\mathcal{J}^{\alpha} = \mathcal{J} - \alpha \mathbb{E}_{\tilde{p}_l(\boldsymbol{x},y)}[-\log q_\phi(y|\boldsymbol{x})] \tag{3.20}$$

Donde α restringe el peso que tiene el segundo término a la hora de aprender. Siguiendo la sugerencia del autor, hemos utilizado $\alpha = 0{,}1N$ en los experimentos.

3.5. Truco de reparametrización

En las secciones anteriores, hemos visto como nuestras funciones objetivo 3.15, para el modelo M1, y 3.20 en el caso del M2, requerían del cálculo de esperanzas respecto a la aproximación del posterior $q_\phi()$. El truco de reparametrización nos permite estimar la esperanza de una forma que permite el cálculo de los gradientes, utilizados para optimizar la función objetivo, gracias a la generación de variables latentes a partir de las distribuciones de los *encoder*.

La idea fundamental es [14]: sea z una variable continua aleatoria y $\tilde{\boldsymbol{z}} \sim q_\phi(\boldsymbol{\epsilon}, \boldsymbol{x})$ una distribución condicionada, $\boldsymbol{z}$ se puede expresar como una variable determinista $\boldsymbol{z} = g_\phi(\boldsymbol{\epsilon}, \boldsymbol{x})$, donde $\boldsymbol{\epsilon}$ es una variable auxiliar con una verosimilitud marginal $p(\boldsymbol{\epsilon})$ y $g_\phi()$ es una función parametrizada por $\boldsymbol{\phi}$.

Mientras que el método propuesto permite generar muestras de distintos tipos de distribuciones, nosotros vamos a centrarnos en la que modela las variables latentes de nuestro modelo, es decir, una distribución Gaussiana $z \sim p(\boldsymbol{z}|x) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2))$. De esta forma, utilizaremos la ecuación 3.21 para obtener muestras de las variables latentes del modelo M1:

$$z = \boldsymbol{\mu}(\boldsymbol{x}) + \boldsymbol{\sigma}(\boldsymbol{x})\boldsymbol{\epsilon} \tag{3.21}$$

Y 3.22 en el modelo M2:

$$z = \boldsymbol{\mu}(\boldsymbol{x}, \boldsymbol{y}) + \boldsymbol{\sigma}(\boldsymbol{x})\boldsymbol{\epsilon} \tag{3.22}$$

En ambos casos ϵ es una variable auxiliar de ruido $\epsilon \sim \mathcal{N}(0, 1)$

4

IMPLEMENTACIÓN

4.1. Conjunto de datos MNIST

Debido a que nuestros modelos utilizan las distribuciones de probabilidad de los datos, es necesario conocer cual es el input para poder comprender la elección de una distribución u otra. Nosotros, para los experimentos hemos utilizado el conjunto de datos MNIST. Este dataset contiene imágenes de dígitos (del 0 al 9) escritos a mano. Las imágenes son de 28x28 píxeles. En la figura 5.1(b), tenemos una pequeña muestra del data set. Para cargarlo hemos utilizado la función de TensorFlow `read_datasets`, que nos devuelve un objeto `Datasets` que contiene tres subconjuntos del MNIST original, en los que diferenciamos entre las imágenes propiamente dichas y sus etiquetas. Las imágenes están codificadas como arrays de NumPy de 784 elementos (la imagen original aplanada) en las que cada elemento se corresponde con un píxel de la imagen original. El valor de cada uno de los pixeles está comprendido en el intervalo $\{x \in \mathbb{R} : 0 \leq x \geq 1\}$. De esta forma, el 0 representa un píxel que no está coloreado y el 1, coloración total, pudiendo tener valores de intensidad entre el 0 y el 1.

Además de la imagen como tal, para cada uno de los puntos tenemos disponible también la información de la clase a la que pertenece el dígito en codificación *one hot*. Como solo hay 10 clases (dígitos del 0 al 9), utilizaremos para representarlas un NumPy array de 10 elementos. De esta forma si el punto representa un '1', en el array de las etiquetas solo estará activo el segundo elemento (0,1,0,0,0,0,0,0,0,0).

Los tres subconjuntos en los que está divido el conjunto original y el número de puntos que contiene cada uno están presentes en la figura 4.1:

Train	Validation	Test
55000	5000	10000

Tabla 4.1: Subconjuntos de MNIST cuando leemos el dataset con la función `read_datasets` y el número de datos de cada uno.

4.2. Implementación del modelo discriminatorio de características latentes

Todos los modelos se han implementado utilizando Python 3 y TensorFlow 1.13.1. En concreto, en esta sección vamos a detallar los pasos seguidos para implementar el modelo M1 planteado en 3.2. Comenzaremos con los datos de entrada del modelo para luego continuar con la generación de las variables latentes utilizando el encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ 3.4 y su posterior decodificación con el modelo generativo $p_\theta(\boldsymbol{x}|\boldsymbol{z})\ p(\boldsymbol{z})$ 3.3. Terminaremos con el cálculo de $\tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x})$ de la sección 3.4.1 y la optimización del mismo.

4.2.1. Datos de entrada

Utilizaremos para entrenar el modelo el conjunto de datos MNIST que hemos detallado en la sección 4.1. Como ya explicamos en la sección 3.2, el modelo M1 se enmarca dentro del aprendizaje no supervisado, por lo que desechamos las etiquetas de la clase de los puntos. Además, para aumentar el número de puntos de entrenamiento y por ende, mejorar los resultados del modelo, apilamos las imágenes de los subconjuntos 'train' y 'validation' para obtener un nuevo subconjunto de 60000 datos.

En TensorFlow los datos se introducen mediante el uso de `placeholder`, por lo que tenemos que definir uno con dimensiones `(None, 784)`, siendo el primer elemento de la dimensionalidad el número de datos de cada minibatch, que puede ser variable cuando el número total de puntos no es divisible entre el tamaño del minibatch, de modo que lo incializamos a `None`. El segundo argumento es el número de elementos que tiene cada punto, en este caso 784 para las imágenes de 28x28 píxeles. Por otra parte, el tipo de datos de los placeholder es `float32`, para los valores reales de la intensidad de coloración del pixel.

Una vez que hemos definido el método que calcula la función objetivo, que depende de los puntos de entrada ($\mathcal{L}(\theta, \phi; \boldsymbol{X})$), podemos evaluar el `Tensor` con su resultado introduciendo mini batches en el `placeholder` (ya que en la ecuación 3.16 hemos estipulado hacerlo de este modo). Sin embargo, antes tomamos una permutación de nuestros datos apilados para asegurar que tenemos las máximas representaciones posibles de todas las clases. Además, normalizamos los datos como probabilidades de Bernoulli, es decir, los píxeles solo pueden tomar los valores 0 o 1.

Por otra parte, como el número de puntos del mini batch puede ser variable, hemos definido otro `placeholder` para este valor, que será también un argumento del método que calcula el Lower Bound del mini batch. En su caso, nos bastó con definir el tipo `float32`. Escogimos este tipo para no tener problemas a la hora de multiplicarlo por el valor del Lower Bound para calcular la aportación del mini batch.

4.2.2. Cálculo de los parámetros del encoder

En esta sección vamos a explicar la implementación de la red neuronal que genera los parámetros del encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$, definida en la ecuación 3.4, $\boldsymbol{\mu}_\phi(\boldsymbol{x})$ y $\sigma_\phi(\boldsymbol{x})$. En primer lugar vamos a mostrar la topología de la red, disponible en el fragmento de código que aparece en la sección:

Código 4.1: Red neuronal para calcular los parámetros de la distribución del encoder del modelo M1.

```
1  def deepnn_encoder(x, num_z, num_hidden_neurons=600):
2      """
3      Computes the mean and log variance of the encoder distribution using two hidden
4      layers.
5      :param x: numpy array with the original image normalized with binarization
6      :param num_z: number of latent variables
7      :param num_hidden_neurons: number of neurons in the hidden layers
8      :return: mean and log of the variance of the encoder distribution
9      """
10     h1 = tf.contrib.layers.fully_connected(x, num_hidden_neurons, activation_fn=tf.nn.softplus,
11                                            scope='encoder_1', reuse=tf.AUTO_REUSE)
12     h2 = tf.contrib.layers.fully_connected(h1, num_hidden_neurons, activation_fn=tf.nn.softplus,
13                                            scope='encoder_2', reuse=tf.AUTO_REUSE)
14     mu = tf.contrib.layers.fully_connected(h2, num_z, activation_fn=None,
15                                            scope='encoder_mu', reuse=tf.AUTO_REUSE)
16     logvar = tf.contrib.layers.fully_connected(h2, num_z, activation_fn=None,
17                                                scope='encoder_logvar', reuse=tf.AUTO_REUSE)
18     return mu, logvar
```

Como vemos, hemos basado la implementación de la red en el método `fully_connected` de TensorFlow. Este método calcula la salida de una capa de una red neuronal, es decir, la multiplicación entre el *input* de la red (el primer argumento) y los pesos de cada una de las conexiones con las neuronas ocultas, cuyo número está definido en el segundo argumento. La función de activación se define en `activation_fn`. Por otro lado, hay dos argumentos extra, `scope` y `reuse`, que nos permiten automatizar la creación de los pesos, de modo que TensorFlow se encarga de crear sus variables asociadas en el ámbito definido en `scope` y de reutilizarlas (además de crearlos cuando no existan en ese ámbito). Estas dos opciones son muy útiles cuando la red se utiliza en varias partes del código, ya que si no se indican, cada vez que se invoque se utilizarán nuevos pesos. En consecuencia, se ha nombrado el `scope` de cada capa de forma distinta para que no surjan colisiones.

Respecto a los datos de entrada de la red, estos son siempre la salida de la capa anterior, menos en la primera capa (línea 10) en la que los datos de entrada son las imágenes binarizadas. Cabe destacar que las capas que calculan la media y el logaritmo de la varianza tienen como entrada la misma capa intermedia '`h2`'.

Para construir las dos capas ocultas (lineas 10 y 12), hemos utilizado 600 neuronas y la función de activación `softplus` (también de la librería de TensorFlow). Nos decantamos por esa función

siguiendo la recomendación del autor en [6] y tras contrastar que era la que mejor resultados arrojaba (en el modelo M2 esta diferencia se hizo mucho más acusada).

Por último, nos queda hablar de las capas que generan los parámetros de la distribución (líneas 14 y 16). Ambas tienen como dimensión de las neuronas de salida el número de variables latentes que queremos generar por punto: **50**. En la literatura que hemos encontrado sobre el tema era recurrente este valor. Por otra parte, la función de activación es lineal (codificado como `None` en TensorFlow), ya que la media no está restringida en ningún rango de valores y el logaritmo de la varianza tampoco. Cuando queramos acceder al valor de la varianza, simplemente tendremos que calcular $\exp(\texttt{logvar})$.

4.2.3. Muestreo de variables latentes

Una vez que disponemos de los parámetros de la distribución del *encoder* $\boldsymbol{\mu}_\phi(\boldsymbol{x})$ y $\boldsymbol{\sigma}_\phi(\boldsymbol{x})$, podemos hacer uso del truco de reparametrización definido en la sección 3.5 para generar muestras de las variables latentes de cada punto original. En el caso del modelo M1, obtenemos las variables latentes a través de $\boldsymbol{z} = \boldsymbol{\mu}_\phi(\boldsymbol{x}) + \boldsymbol{\sigma}_\phi(\boldsymbol{x})\boldsymbol{\epsilon}$. Podemos ver este proceso en el método `reparamerization_trick` del código:

Código 4.2: Método que aplica el truco de reparametrización

```
1  def reparamerization_trick(mu, logvar, dim_z, batch_size):
2      """
3      This applies the reparametrization trick
4      :param mu: mean of each latent variable
5      :param logvar: log of the variance of the latent variables
6      :param dim_z: number of latent variables to sample
7      :param batch_size: number of points to sample
8      :return: array of dimensions (batch_size, dim_z) with the latent variables sampled using mu and logvar
9      """
10     return tf.sqrt(tf.exp(logvar)) *tf.random_normal(shape=[batch_size, dim_z], dtype=tf.float32) + mu
```

En primer lugar, generamos ruido a partir de una distribución Gaussiana con media 0 y desviación típica 1 utilizando `random_normal`. Este método crea un array con dimensión `(batch_size, dim_z)` (correspondiente con el número de puntos y el número de variables latentes de cada punto) de valores generados a partir de una distribución normal estándar. Una vez que hemos generado el ruido, simplemente calculamos el valor de la ecuación 3.21 (línea 10). Para ello utilizamos la media y el logaritmo de la varianza generados en la red presentada en la sección 4.2.2. Cabe destacar que la ecuación no utilza $\log \boldsymbol{\sigma}^2$ (que es lo que representa `logvar`), sino la desviación típica. Para obtenerla calculamos $\sqrt{e^{\texttt{logvar}}}$.

4.2.4. Cálculo de los parámetros del decoder

Posteriormente al muestreo de variables latentes, podemos calcular los parámetros de la distribución del modelo generativo $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, definido en la sección 3.2. Lo primero que tenemos que hacer es aclarar qué distribución utilizaremos para la verosimilitud $f(\boldsymbol{x};\boldsymbol{z},\boldsymbol{\theta})$. En la sección 4.2.1, hemos mencionado que normalizamos los datos como probabilidades de Bernouilli para el entrenamiento. De esta forma, calcularemos $p_\theta(\boldsymbol{x}|\boldsymbol{z})$ como una distribución de Bernoulli. Sin embargo, nuestra red, en lugar de calcular la probabilidad de cada píxel, generará su *logit*, de modo que para obtener el valor de esta, tendremos que transformar el logit a través de la sigmoidie.

Para construir la topología hemos utilizado `fully_connected` de nuevo. El input de la red son las variables latentes generadas mediante el truco de reparametrización. Para configurar las dos capas ocultas hemos fijado el número de neuronas en 600 y hemos escogido `softplus` como función de activación. Por último, la capa de salida tiene función de activación lineal para generar los *logits*. En concreto, genera `dim_x` (784) logits por punto del minibatch, relativos a los píxeles que forman cada imagen. Los parámetros `scope` y `reuse` tienen el mismo objetivo que en la red del decoder (sección 4.2.2).

4.2.5. Cálculo de la función objetivo

En esta sección vamos a detallar el cálculo del $\mathcal{L}$ del modelo M1, definido en la sección 3.4.1. Vamos a recordar la ecuación 3.12 para mayor comodidad a la hora de comprender los cálculos que se llevan a cabo:

$$\tilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}^{(i)}) = -KL(q_\phi(\boldsymbol{z}|\boldsymbol{x}^{(i)})||p_\theta(\boldsymbol{z})) + \frac{1}{L}\sum_{l=1}^{L}\log p_\theta(\boldsymbol{x}^{(i)}|\boldsymbol{z}^{(i,l)})$$

Este es el valor del Lower Bound para cada punto del conjunto de datos. En el método `calculate_LM1` del fragmento de código que aparece en la sección calculamos este valor por cada punto del mini batch y ponderamos su aportación al Lower Bound del dataset completo.

Comenzaremos explicando como obtenemos el segundo término de la ecuación, es decir, la esperanza $\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x}^{(i)})}[\log p_\theta(\boldsymbol{x}^{(i)}|\boldsymbol{z})]$. Como mencionamos en la sección 3.4.1, vamos a hacer una estimación por muestreo, de modo que el método para calcularlo (ecuación 3.13), requería generar L samples distintos de $\boldsymbol{z}$ para promediar el valor de $\log p_\theta(\boldsymbol{x}^{(i)}|\boldsymbol{z})$. Sin embargo, tanto la literatura sobre el tema como nuestros experimentos, coincidían en que era suficiente con utilizar $L = 1$ (generar una sola muestra de variables latentes por punto) siempre que el tamaño del mini batch fuese al menos de 100, que es el mínimo que hemos escogido en nuestros experimentos. De este modo, el cálculo del segundo término requiere generar muestras de las variables latentes con el método explicado en la sección 4.2.3, para obtener, a través de `deepnn_decoder` (sección 4.2.4), los *logits* de la distribución

Código 4.3: Método que calcula la aportación del $\mathcal{L}$ por cada mini batch en el modelo M1

```
1  def calculate_LM1(x, dim_z1, minibatch_size):
2      """
3      Computes the mini batch Lower Bound
4      :param x: mini batch data
5      :param dim_z1: number of latent variables per data point
6      :minibatch_size: number of points in the mini batch
7      :return: the mini batch LB and the logits of the decoder
8      """
9      z1, mu, logvar = generate_z(x, dim_z1)
10     output_decoder = deepnn_decoder(z1, 784)
11     logpxz1 = tf.nn.sigmoid_cross_entropy_with_logits(labels=x, logits=output_decoder)
12     esp = tf.reduce_sum(logpxz1)
13     kl = -0.5 *tf.reduce_sum(1. + logvar -tf.square(mu) -tf.exp(logvar))
14
15     return (tf.dtypes.cast(minibatch_size, tf.float32) / 60000.) *(kl + esp), output_decoder
```

$p_\theta(\boldsymbol{x}|\boldsymbol{z})$. Ya hemos mencionado que estos *logits* son el inverso de la sigmoide de las probabilidades de la distribución de Bernoulli correspondiente. En este punto, podemos hacer uso del método de TensorFlow `sigmoid_cross_entropy_with_logits`, introduciendo la salida de la red que calcula los parámetros del *decoder* y los $\boldsymbol{x}$ originales para calcular su valor por píxel y punto, ya que el método realiza los cálculos equivalentes: calcular la probabilidad pasando los *logits* por la sigmoide (de modo que ya tenemos la probabilidad de Bernoulli del píxel), aplicar la fórmula de su cálculo, y obtener el logaritmo del mismo. Podemos ver primero la generación de *logits* en la línea 10 y la aportación al $\mathcal{L}$ del mini batch de $\log p_\theta(\boldsymbol{x}|\boldsymbol{z})$ en la línea 11.

Para calcular la aportación del término KL recurrimos a la solución planteada en la ecuación 3.14. Esta solución requiere resolver $\frac{1}{2}\sum_{j=1}^{J}(1+\log\sigma_j^2(\boldsymbol{x})-\mu_j^2(\boldsymbol{x})-\sigma_j^2(\boldsymbol{x}))$. De esta forma, necesitamos la media y el logaritmo de la varianza que hemos obtenido con la red `deepnn_encoder` de la sección 4.2.2 (línea 9) y obtener la varianza transformando su logaritmo. El cálculo final solo requiere aplicar la fórmula.

Por último sumamos la aportación de cada término y la promediamos por el tamaño del mini batch. Cabe destacar que el valor de retorno es el negativo del Lower Bound, en la siguiente sección explicamos por qué es recomendable mantener su valor negativo.

4.2.6. Bucle de entrenamiento y optimización

Una vez que hemos calculado el valor de nuestra función objetivo, tenemos que escoger un método para su optimización. Aunque en el artículo proponen utilizar RMSprop o AdaGrad, nosotros hemos optado por utilizar Adam, otro método de optimización estocástica con mejores propiedades de convergencia, sobre todo para entrenar MLP (como son nuestras redes que generan los parámetros de

las distribuciones) [17]. Por otra parte, este método nació de las necesidades del AEVB planteado por Kingma en el artículo en el que basamos la implementación de nuestro modelo. Además, ya existía una implementación de Adam en TensorFlow, lo que hacía que se perfilase como la mejor alternativa posible. El único punto negativo es que este método solo puede minimizar la función objetivo, mientras que nuestra meta, es maximizar el $\mathcal{L}$. Es por eso, que en la sección 4.2.5 hemos mencionado que aunque el valor que calculamos se corresponde con $-\mathcal{L}$, esto no es erróneo, ya que al minimizar el negativo del Lower Bound, realmente estamos maximizándolo. Podemos ver esto en el siguiente fragmento de código:

Código 4.4: Definición del método de optimización del $\mathcal{L}$ y selección de *learning rate* en el modelo M1

```
1  # Placeholders
2  x = tf.placeholder(tf.float32, [None, dim_x])
3  size_mini_batch = tf.placeholder(tf.float32)
4
5  lowerbound_M1, gen_logits = calculate_LM1(x, dim_z1, size_mini_batch)
6  train_step = tf.train.AdamOptimizer(1e-3).minimize(lowerbound_M1)
```

En concreto, en el código 4.4 vemos la definición de los `placeholder` de los que hemos hablado en la sección 4.2.1: el de los datos de entrada (línea 2), y el del tamaño del mini batch (línea 3). Posteriormente, calculamos el valor del Lower Bound (línea 5) utilizando la función `calculate_LM1`, que implementa el método descrito la sección 4.2.5, a la que pasamos los datos de entrada y el tamaño del mini batch mediante los `placeholder`, y la dimensión de las variables latentes, que como ya hemos mencionado, es 50. Por último, nos queda la definición del método de optimización Adam, para lo que utilizamos la implementación de TensorFlow `AdamOptimizer`, que recibe como argumento la *learning rate*, para la que hemos utilizado un valor de $1e-3$ siguiendo la recomendación del autor del artículo. Por otra parte, fijamos el valor a minimizar como el negativo del $\mathcal{L}$ que hemos calculado, lo que es equivalente a maximizar el Lower Bound. Este método devuelve un grafo de operaciones para actualizar las variables de los pesos de modo que minimicemos el valor de la función objetivo.

Vamos a dedicar este párrafo a arrojar algo más de luz sobre la implementación del bucle de entrenamiento, basada en el uso de dos bucles anidados: el primero, sobre las épocas establecidas, y el segundo, para calcular el valor la aportación $\mathcal{L}$ por cada mini batch. Como vemos en el fragmento de código 4.5, en el bucle sobre el número de épocas (línea 1) solo incializamos el valor de una variable auxiliar para almacenar el valor del Lower Bound calculado para las pruebas y creamos la permutación, que nos devuelve un array con los índices de las imágenes desordenadas. El segundo bucle (línea 5) itera sobre el número de mini batches en los que se dividen los puntos originales y en él evaluamos el valor del `Tensor` con el resultado del Lower Bound (línea 13) del mini batch binarizado (línea 8) permutado con los índices obtenidos, y aplicamos el `train_step`, correspondiente con el grafo de operaciones necesarias para actualizar θ y ϕ, definido como resultado de aplicar Adam sobre el $\mathcal{L}$

utilizando `run` (línea 12).

Código 4.5: Bucle de entrenamiento del modelo M1

```
1   for epoch in range(n_epoch):
2       lowerBound_train = 0
3       perm = np.mod(np.random.permutation(np.maximum(n_batch *size_batch, n_train)),
            n_train)
4
5       for i in range(int(n_train / n_batch)):
6           indices = perm[(i *size_batch): np.minimum((i + 1) *size_batch, len(perm))]
7           # Apply binarization
8           x_batch = (np.random.uniform(size=(len(indices), data_labeled.shape[1])) <
9                      data_labeled[indices,
10                     :]).astype(np.float32)
11
12          train_step.run(feed_dict={x: x_batch, size_mini_batch: x_batch.shape[0]})
13          lowerBound_train += -1. *lowerbound_M1.eval(
14              feed_dict={x: x_batch, size_mini_batch: x_batch.shape[0]})
```

Por otra parte, cabe destacar que todas las evaluaciones de los `Tensor`, es decir el bucle de entrenamiento, se ejecutan dentro del contexto `Session` de TensorFlow.

4.3. Implementación del modelo generativo semi supervisado

Para describir la implementación del modelo generativo de aprendizaje semi supervisado (M2) vamos a seguir el mismo enfoque que en el caso del modelo M1. Comenzaremos detallando los datos de entrada del modelo, para seguir con la implementación de la red que genera los parámetros del *encoder* y la del modelo generativo. Por último mostraremos como hemos calculado el Lower Bound y su optimización.

4.3.1. Datos de entrada

Utilizamos el conjunto de datos MNIST para entrenar y probar el modelo. Como en el caso del modelo M1, apilamos las imágenes de los subconjuntos 'train' y 'validation' con el fin de aumentar el número de puntos disponibles, de modo que el resultado es un nuevo subconjunto del MNIST con 60000 puntos. Sin embargo, en este caso si que utilizamos parte de las etiquetas de las imágenes de ejemplo, pero nunca su totalidad. En la sección 5 mostramos los valores utilizados en los distintos experimentos. Debido a esta casuística, los dos subconjuntos son:

- Etiquetado: una parte del conjunto de datos tiene etiquetas, por lo que deberemos crear

dos `placeholder` distintos para este subconjunto, uno para las imágenes, con dimensión `(None, 784)`, y otro para las clases, de dimensión `(None, 10)`. El primer elemento de la dimensión es `None` por la misma razón que en el caso del `placeholder` del modelo M1 (sección 4.2.1). En el caso del `placeholder` para las imágenes, el segundo elemento se corresponde con los 784 píxeles que conforman la imagen, mientras que el de las etiquetas es 10, porque es el número de elementos necesarios para expresar las 10 clases posibles en codificación *one hot*.

- No etiquetado: el subconjunto del MNIST que no tiene etiquetas. Como solo utilizamos las imágenes, las dimensiones de su `placeholder` se fijan a `(None, 784)`.

Además, con el fin de probar el funcionamiento de la red, añadimos otro `placeholder` para el subconjunto de test con las imágenes, con la misma dimensionalidad que la utilizada para el subconjunto no etiquetado. En todos los casos hemos definido el tipo `float32`.

Cabe destacar que en este caso el número de mini batches por época viene definido por el número de datos etiquetados, de modo que por cada iteración solo hay un dato etiquetado y $\frac{\texttt{num\_label}}{\texttt{num\_batch}}$ no etiquetados.

De nuevo, como en el caso del M1, permutamos los índices para aleatorizar los puntos de cada mini batch y binarizamos las imágenes antes de introducirlos en la función que calcula el Lower Bound, de modo que los píxeles toman solo como valor 1 o 0.

4.3.2. Cálculo de los parámetros del encoder

El objetivo de este método consiste en calcular los parámetros del encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x}, y)$ (ecuación 3.7), $\mu_\phi(\boldsymbol{x}, y)$ y $\sigma^2_\phi(\boldsymbol{x})$. Recalcamos el hecho de que la media depende de las etiquetas además de las imágenes, mientras que la varianza solo de las últimas. En consecuencia, necesitaremos una red para la media y otra para la varianza. Respecto al número de capas ocultas, en un primer momento probamos a utilizar dos, como en el M1, pero los mejores resultados los obtuvimos con solo una capa. Teniendo esto en cuenta, vamos a comenzar a describir la implementación de las dos redes que encontramos en el fragmento de código 4.6.

Como vemos, hemos basado la implementación de las capas en el método `fully_connected`. Como ya hemos explicado su uso en la sección 4.2.2, vamos a hablar exclusivamente de los parámetros utilizados con los que las configuramos. Por otra parte, hemos mencionado el hecho de que aunque el método devuelve tanto la media como el logaritmo de la varianza de la distribución, cada resultado se obtiene de una red distinta. Comenzaremos explicando la que calcula $\mu_\phi(\boldsymbol{x}, y)$, que al depender de $\boldsymbol{x}$ e y, su primera capa (línea 12) tiene como entrada el argumento `xy`, que contiene la imagen binarizada y las etiquetas en codificación *one hot*. La función de activación es `softplus` y tiene 500 neuronas en la capa intermedia. Su resultado, `h1_m`, se introduce en la capa que calcu-

Código 4.6: Redes neuronales para calcular los parámetros de la distribución del encoder del modelo M2.

```
def encoder_m2(x, xy, num_z, num_hidden_neurons=500):
    """
    Compute output for the mean and log of the variance of the distribution p(z|x,y).
    The mean depends on x and y but the variance only on x
    :param x: binarized input images
    :param xy: binarized input images and the class label in one hot encoding appended.
    :param num_z: number of latent variables
    :param num_hidden_neurons: number of neurons in the hidden layes
    :return: mean and log of the variance of the distribution p(z|x,y).
    """

    h1_m = tf.contrib.layers.fully_connected(xy, num_hidden_neurons, activation_fn=tf.nn.softplus,
                                             scope='encoder_1_m',
                                             reuse=tf.AUTO_REUSE,
                                             weights_initializer=tf.initializers.truncated_normal(
                                                 stddev=1e-3),
                                             biases_initializer=tf.initializers.zeros)

    h1_v = tf.contrib.layers.fully_connected(x, num_hidden_neurons, activation_fn=tf.nn.softplus,
        scope='encoder_1_v',
                                             reuse=tf.AUTO_REUSE,
                                             weights_initializer=tf.initializers.truncated_normal(
                                                 stddev=1e-3),
                                             biases_initializer=tf.initializers.zeros)

    mean = tf.contrib.layers.fully_connected(h1_m, num_z, activation_fn=None,
        scope='output_encoder_m',
                                             reuse=tf.AUTO_REUSE,
                                             weights_initializer=tf.initializers.truncated_normal(
                                                 stddev=1e-3),
                                             biases_initializer=tf.initializers.zeros)

    logvar = tf.contrib.layers.fully_connected(h1_v, num_z, activation_fn=None,
        scope='output_encoder_v',
                                               reuse=tf.AUTO_REUSE,
                                               weights_initializer=tf.initializers.truncated_normal(
                                                   stddev=1e-3),
                                               biases_initializer=tf.initializers.zeros)

    return mean, logvar
```

la la media (línea 25). Esta capa genera las medias de `num_z` variables latentes y tiene función de activación lineal. El número de variables latentes utilizado ha sido 50.

La red que calcula la varianza tiene la misma topología pero el input de la red, (línea 19) es la matriz con las imágenes (parámetro `x`). La salida vuelve a ser el logaritmo de la varianza (línea 31), de ahí que la función de activación sea lineal.

El uso de los argumentos `reuse` y `scope` tenía el mismo objetivo que en el resto de redes, permitir que TensorFlow se encargue de la creación de las variables de los pesos y no crear distintas instancias de las redes cada vez que se llaman.

Por otra parte, hemos inicializado los pesos de las redes utilizando `truncated_normal`, es decir, a partir de una distribución Gaussiana con media 0 y desviación típica 1e-3. Como peculiaridad, todos los valores generados por este método cuya magnitud sea dos veces σ son desechados. Los pesos de las conexiones con los bias se han incializado a 0.

4.3.3. Muestreo de variables latentes

Para generar las variables latentes del modelo M2 hacemos uso del truco de reparametrización, en concreto, de la ecuación 3.22. El método es el mismo que el utilizado en el modelo M1, con la diferencia de que en este caso la media depende tanto de $\boldsymbol{x}$ como de $\boldsymbol{y}$, pero la generación de ruido, y el proceso por el que obtener la desviación típica a partir del logaritmo de la varianza, son los mismos que los explicados en la sección 4.2.3.

4.3.4. Cálculo de los parámetros del decoder

El método descrito en esta sección se encarga de calcular los parámetros de la distribución del *decoder* $p_\theta(\boldsymbol{x}|y,\boldsymbol{z})$. Como hemos normalizado los datos utilizando binarización, calcularemos la función de verosimilitud $f(\boldsymbol{x};y,\boldsymbol{z},\boldsymbol{\theta})$ como una distribución de Bernoulli. De esta forma, la red generará las probabilidades de los puntos $\boldsymbol{x}$ dada la clase y las variables latentes. Sin embargo, como en el M1, por motivos de rendimiento, optamos por generar los logits. De esta forma, la red queda formada por una capa intermedia con 500 neuronas y función de activación `softplus`, y otra de salida, que genera `dim_x` logits, correspondientes a las probabilidades de los 784 píxeles de las imágenes originales. La entrada de la red son las variables latentes $\boldsymbol{z}$ y las etiquetas y concatenadas. Los pesos para las conexiones entre las neuronas se han inicializado de nuevo a partir de una Gaussiana con media 0 y desviación típica 1e-3 utilizando `truncated_normal`. Las conexiones con los bias fueron inicializadas a 0.

4.3.5. Cálculo de los parámetros del clasificador

El clasificador inferido $q_\phi(y|\boldsymbol{x})$ es el encargado de hacer las predicciones para las nuevas imágenes. Tiene la forma de un clasificador discriminante y viene dado por una distribución multinomial $Cat(y, \boldsymbol{\pi}(\boldsymbol{x}))$. Las posibles categorías se corresponden con las 10 clases de dígitos escritos. Como en el resto de distribuciones, hemos utilizado un MLP para calcular el valor de sus parámetros. En este caso, al tratarse de una distribución multinomial, el resultado de la red es un array de `num_clases` elementos, cada uno de los cuales se corresponde con los logits de la probabilidad de la clase. Hacemos especial hincapié en remarcar el hecho de que la salida de la red no es la probabilidad de cada clase como tal, sino un array con valores reales al que aplicaremos la función `softmax` para normalizar y así obtener las verdaderas probabilidades. En consecuencia, la función de activación de la última capa es lineal. Aunque podríamos haber utilizado `softmax` para esto, vimos que obtenemos mejores resultados con la salida lineal y aplicando a posteriori `softmax`. El resultado es un mapeo con el array de clases en formato *one hot*, es decir, el primer elemento de la salida se corresponde con el logit de la probabilidad de la clase '0', y el último, con el de la clase '9'.

La topología de la red está compuesta por dos capas, para las que utilizamos `fully_connected`. El input de la red, se corresponde con las imágenes binarizadas. Esta capa tiene 200 neuronas ocultas y función de activación `softplus`. Su resultado se introduce en la capa de salida, que calcula los logits por cada punto del mini batch, a los que posteriormente tenemos que aplicar `softmax` para obtener las probabilidades que se utilizan como parámetro $\boldsymbol{\pi}(\boldsymbol{x})$ de la distribución multinomial de $q_\phi(y|\boldsymbol{x})$.

De nuevo, los pesos de las conexiones entre las neuronas se han inicializado a partir de una Gaussiana con media 0 y desviación típica 0.001, mientras que el valor inicial de las conexiones con los bias es 0. En este caso también hemos utilizado los parámetros `reuse` y `scope` para delegar en TensorFlow el manejo de los pesos.

4.3.6. Cálculo del Lower Bound de los datos etiquetados

En la ecuación 3.19 definimos el Lower Bound del dataset del modelo M2 como la suma entre el Lower Bound del subconjunto etiquetado $\mathcal{L}(\boldsymbol{x}, y)$ y el no etiquetado $\mathcal{U}(\boldsymbol{x})$. En esta sección explicaremos el cálculo de la componente relativa al subconjunto con etiquetas.

Como en el caso del modelo M1, utilizamos estimaciones mediante el muestro de $\boldsymbol{z}$ para calcular la esperanza que conforma el Lower Bound (ecuación 3.17) de los datos etiquetados. De nuevo, siguiendo las sugerencias del autor del artículo, hemos optado por generar un solo vector de variables latentes por cada punto original. Vemos la implementación del método que calcula su valor en el fragmento de código presentado en esta sección.

Debido a que la media del *encoder* de este modelo depende tanto de $\boldsymbol{x}$ como de y, tenemos que

Código 4.7: En este fragmento de código se muestra el método que calcula el Lower Bound del subconjunto de datos etiquetados.

```
1  def compute_L_x_y(x, y, latent_dims, pi_labeled):
2
3      xy = tf.concat([x, y], 1)
4      mean_z, log_var_z = encoder_m2(x, xy, latent_dims)
5      z = reparamerization_trick(mean_z, log_var_z, latent_dims, tf.shape(x)[0])
6
7      zy = tf.concat([z, y], 1)
8      logits = decoder_m2(zy, x.shape[1].value)
9
10     log_p_x_zy = tf.reduce_sum(-1.0 *tf.nn.sigmoid_cross_entropy_with_logits(logits=logits,
           labels=x), axis=1)
11     log_py = tf.reduce_sum(y *tf.log(pi_labeled), axis=1)
12     log_pz = tf.reduce_sum(-0.5 *tf.log(2.0 *np.pi) -0.5 *tf.square(z), axis=1)
13     log_q_z_xy = tf.reduce_sum(-0.5 *tf.log(2.0 *np.pi) -0.5 *log_var_z -0.5, axis=1)
14
15     return log_p_x_zy + log_py + log_pz -log_q_z_xy
```

concatenar a cada vector de la imagen la clase en codificación *one hot* (línea 3). Una vez generada esta nueva matriz, podemos utilizar los MLP presentados en la sección 4.3.2 para obtener $\boldsymbol{\mu}(\boldsymbol{x}, y)$ y $\boldsymbol{\sigma}^2(\boldsymbol{x})$ (línea 4), lo que nos permite generar las muestras de $\boldsymbol{z}$ aplicando el truco de reparametrización (línea 5). Posteriormente, computamos los logits de la distribución de Bernoulli del *encoder* con la red descrita en la sección 4.3.4 (línea 8) pasando las variables latentes y las etiquetas concatenadas (línea 7). En este punto ya tenemos los datos necesarios para estimar $\mathcal{L}(\boldsymbol{x}, y)$.

Vamos a recordar las probabilidades que teníamos que calcular y aprovecharemos para explicar el método utilizado:

- $\log q_\theta(\boldsymbol{z}|\boldsymbol{x}, y)$: ya hemos mencionado que la distribución de las variables latentes es una Gaussiana (ecuación 3.7). Con los parámetros obtenidos de los MLP y las muestras de $\boldsymbol{z}$, podemos calcular ese resultado.
- $\log p_\phi(\boldsymbol{x}|\boldsymbol{z}, y)$: para calcularlo utilizamos de nuevo `sigmoid_cross_entropy_with_logits`, ya que este método nos ofrece una implementación del cálculo del logaritmo de la distribución de Bernoulli utilizando logits.
- $\log p_\theta(y)$: las clases siguen una distribución multinomial, luego, para calcular el logaritmo de la probabilidad de la etiqueta de ese punto, nos basta con multiplicar el parámetro `y` (matriz representando las clases de cada punto) por el logaritmo de `pi_data` (vector con las probabilidades de cada clase), ya que como el único bit que estará activado en `y` es el de la clase actual, al realizar el producto el resultado será el logaritmo de la probabilidad de la etiqueta. Cabe destacar que partimos de la base de que todas etiquetas tienen las mismas probabilidades, es decir, 0.1. De este modo, cuando calculamos $\mathcal{L}(\boldsymbol{x}, y)$, el método `compute_L_x_y`

recibe un array de 10 elementos con valor 0.1 como parámetro `pi_labeled`. Aunque podríamos haber tratado esto como una constante del método, decidimos mantener el parámetro para poder reutilizar la función en el cómputo de $\mathcal{U}(\boldsymbol{x})$, ya que requiere el cálculo de $\mathcal{L}(\boldsymbol{x}, y)$ (ecuación 3.18) y en este caso las probabilidades de cada clase si que difieren entre ellas.

- $\log p(\boldsymbol{z})$: como $p(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ nos basta con calcular su valor utilizando las variables latentes muestreadas como en el fragmento de código.

4.3.7. Cálculo del Lower Bound de los datos no etiquetados

La clave para obtener $\mathcal{U}(\boldsymbol{x})$ (ecuación 3.18) consiste en tratar a las etiquetas como variables latentes sobre las que realizar inferencia. Para ello, calculamos el valor de $\mathcal{L}(\boldsymbol{x}, y)$ para los distintos valores de y posibles y multiplicamos por la probabilidad inferida $q_\phi(y|\boldsymbol{x})$. Posteriormente, al valor del sumatorio, le añadimos la entropía de la distribución. Podemos ver la implementación en el fragmento de código 4.8.

Código 4.8: En este fragmento de código se muestra el método que calcula el Lower Bound del subconjunto de datos no etiquetados.

```
1  def compute_U_x(x, latent_dims, pi_data, n_classes):
2
3      logits_y = inference_clasifier(x, n_classes)
4      prob_y = tf.nn.softmax(logits_y)
5      entropy_y = tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits_y, labels=prob_y)
6
7      sumation = tf.zeros(tf.shape(x)[0])
8
9      for i in range(n_classes):
10         y_label = tf.one_hot(tf.ones(tf.shape(x)[0], dtype=tf.int32) *i, n_classes)
11         sumation += compute_L_x_y(x, y_label, latent_dims, pi_data) *prob_y[:, i]
12
13     return sumation + entropy_y
```

Para implementarlo, comenzamos calculando las probabilidades de las etiquetas aplicando `softmax` a los logits generados a través del MLP descrito en la sección 4.3.5, que recibe como entrada las imagenes binarizadas. Posteriormente, acumulamos el valor de $\mathcal{L}(\boldsymbol{x}, y)$ para cada y recorriendo las 10 clases y pasando en cada iteración un vector representando una etiqueta diferente a `compute_L_x_y`. Además, en cada iteración, multiplicaremos el valor calculado por el método por la probabilidad de la clase actual que hemos generado aplicando `softmax` a los resultados de la red 4.3.5. Una vez que tenemos el sumatorio, solo queda calcular la entropía $\mathcal{H}(q_\phi(y|\boldsymbol{x}))$, para lo que utilizamos el método `softmax_cross_entropy_with_logits_v2`, al que pasamos los logits calculados y las probabilidades.

4.3.8. Cálculo de la función objetivo

En la ecuación 3.19, definimos el Lower Bound del dataset completo como la suma de $\mathcal{L}(\boldsymbol{x}, y)$ y $\mathcal{U}(\boldsymbol{x})$. Sin embargo, para implementar su cálculo utilizando mini batches, tenemos que ponderar la aportación de cada componente multiplicando su valor por el número de datos del subconjunto etiquetado y no etiquetado respectivamente.

A través de la suma de $\mathcal{L}(\boldsymbol{x}, y)$ y $\mathcal{U}(\boldsymbol{x})$ ponderados, hemos obtenido $\mathcal{J}$, pero la función objetivo que debemos optimizar, definida en la ecuación 3.20, contempla un término de regularización $\mathbb{E}_{\tilde{p}_l(\boldsymbol{x},y)}[-\log q_\phi(y|\boldsymbol{x})]$ que permite que el clasificador de inferencia $q_\phi(y|\boldsymbol{x})$ también aprenda de los datos etiquetados. Para calcular esta esperanza primero obtenemos los logits de las probabilidades de la distribución del clasificador a partir de las imágenes del subconjunto de entrenamiento etiquetado. Después utilizamos `softmax_cross_entropy_with_logits_v2`, que convierte los logits en probabilidades a través de `softmax`, para calcular la esperanza de las clases etiquetadas por cada punto. Recordemos que este término tenía un peso de $0{,}1 * N$ sobre $\mathcal{J}$, siendo N el número de imágenes del dataset original.

4.3.9. Bucle de entrenamiento y optimización

Para la optimización de la función objetivo hemos vuelto a utilizar Adam. Sin embargo, en el caso de este modelo no hemos optimizado el Lower Bound $\mathcal{J}$ del conjunto de datos como tal, sino que hemos optado por optimizar su valor multiplicado por el *weight decay*, es decir, el valor de los pesos de las redes que calculan los parámetros de las distribuciones. De este modo, conseguimos que el valor de los pesos no crezca descontroladamente, ya que penalizamos las actualizaciones de pesos que aumentan demasiado su valor. Este término regularizador ha causado un impacto muy positivo en los resultados obtenidos. En la implementación, la aportación de cada red al *weight decay* es la suma del cuadrado del valor de los pesos multiplicado por $0{,}5$. El *weight decay* total es la suma del de cada una de las redes. Para obtener los pesos de las redes hemos recurrido a `get_collection`, que devuelve las variables definidas en un *scope*. De este modo, hemos indicado el *scope* definido en los métodos `fully_connected` que computaban la salida de las capas de las redes para recuperar los pesos.

En este punto podemos utilizar Adam para obtener las operaciones de actualizaciones de pesos que minimizan el negativo del $\mathcal{J}$ regularizado con el *weight decay* (recordemos que debemos utilizar el negativo porque este método no nos permite maximizar la función objetivo, solo minimizarla). Hemos utilizado 3e-4 como *learning rate*.

Respecto al bucle de entrenamiento, es muy similar al descrito en la sección 4.2.6, con la única diferencia de que tenemos que generar dos permutaciones, una para el subconjunto etiquetado y otra para el no etiquetado. Una vez que tenemos los índices permutados, por cada época, recorremos el

conjunto de datos entero utilizando mini batches y aplicamos el grafo de operaciones de optimización de los pesos que devuelve `AdamOptimizer` al evaluarlo poblando los `placeholders` definidos en la sección 4.3.1 con los mini batches de datos etiquetados y no etiquetados (con los píxeles de las imágenes binarizados). Es decir, las actualizaciones de los pesos se aplican por cada mini batch.

4.4. Modelo generativo semi supervisado apilado

Este modelo utiliza las variables latentes generadas por el M1 para entrenar el M2, en lugar de utilizar las imágenes. Igualmente, para realizar tareas de clasificación, primero obtendremos las variables latentes con M1 y clasificaremos introduciéndolas en M2. De este modo, la distribución del *encoder* del modelo M2 $p_\theta(\boldsymbol{x}|\boldsymbol{z}, y)$ ya no es una distribución de Bernoulli, sino Gaussiana, con parámetros $\mu_\theta(\boldsymbol{z1}, y)$ y $\sigma_\theta(\boldsymbol{z1}, y)$, donde $\boldsymbol{z1}$ son las variables latentes generadas por el modelo M1. Este cambio se debe a que antes la entrada de la red era la probabilidad de activación de los píxeles, que en consecuencia, seguían una distribución de Bernoulli, mientras que ahora son datos reales, las variables latentes, que siguen una distribución normal. Destacamos el hecho de que la entrada de la red que genera ambos parámetros del *encoder* es la misma, $\boldsymbol{z1}$ e y concatenados. Del mismo modo, cuando calculemos $\mathcal{L}(\boldsymbol{x}, y)$ (ecuación 3.18), tenemos que tener esto en cuenta para obtener $\log p_\theta(\boldsymbol{x}|y, \boldsymbol{z})$ y aplicar la fórmula de la distribución normal en lugar de la de Bernoulli. Para implementar este modelo nos basamos en el M1 y M2 que ya habíamos construido.

El primer paso era conseguir codificar el dataset MNIST original como variables latentes $\boldsymbol{z1}$ de dimensión 50. Para ello simplemente entrenamos el modelo M1 durante 200 épocas y, una vez finalizado el proceso de aprendizaje, generamos los parámetros de las variables latentes de los subconjuntos de train y validación concatenados y el de los datos de test, y los volcamos a un csv. Ahora que tenemos disponible los parámetros de la distribución de las variables, leemos el contenido de los ficheros con ellas desde el modelo M2 para utilizarlos como entrada. Cabe destacar, que aunque tenemos disponibles tanto la media como la varianza, **utilizamos solamente la media para las tareas de entrenamiento y clasificación**.

Para adaptar el modelo M2, lo único que tuvimos que considerar fue el cambio de distribución del *encoder* que ya hemos comentado al comienzo de la sección. Del mismo modo, los `placeholder` para las imágenes de los subconjuntos etiquetados y no etiquetados ahora se utilizan para cargar las variables latentes, por lo que su dimensión debe ser de 50 elementos por punto, en lugar de los 784 de las imágenes. Una vez que hemos modificado el método para calcular su valor y adaptado los `placeholder` a la dimensión de las variables latentes, el resto del modelo es completamente igual.

5

Pruebas y resultados

5.1. Comparativa de nuestros modelos con el estado del arte

En esta sección vamos a comparar los resultados arrojados por nuestros modelos con los de otras técnicas del estado del arte. Para entrenar los modelos hemos utilizado 100, 600 y 3000 datos etiquetados. Por otra parte, como el modelo M1 no es capaz de hacer predicciones por si mismo, hemos decidido utilizar un MLP para llevar a cabo la tarea de clasificación a partir de las variables latentes generadas por el mismo, en concreto, la implementación de Scikit-Learn `MLPClassifier`. Debemos mencionar también que para entrenar los modelos M2 y M1+M2 utilizamos dos modos distintos de seleccionar los mini batches. En el caso del M2, para cada época, el número de mini batches viene dado por el número de datos etiquetados. Como no fuimos capaces de obtener muy buenos resultados con esta aproximación en el modelo M1+M2, optamos por utilizar 100 mini batches para el entrenamiento cuando tenemos 100 etiquetas, y 200 en los otros dos casos, igual que hace Kingma en su implementación, mencionada en su artículo [6].

					Nuestra implementación			**Implementación Kingma**	
N	CNN	TSVM	CAE	Associative CNN	NN + M1	M2	M1+M2	M2	M1+M2
100	22.98	16.81	13.47	11 ($\pm$ 0.08)	39.4	10.31 ($\pm$ 4.06)	4.04 ($\pm$ 0.06)	11.97 ($\pm$ 1.71)	3.33 ($\pm$ 0.14)
600	7.68	6.16	6.3	-	15.24	4.096 ($\pm$ 0.58)	3.676 ($\pm$ 0.11)	4.94 ($\pm$ 0.13)	2.59 ($\pm$ 0.05)
3000	3.35	3.45	3.32	-	7.7	4.09 ($\pm$ 0.51)	3.645 ($\pm$0.14)	3.92 ($\pm$ 0.63)	2.18 ($\pm$ 0.04)

Tabla 5.1: Resultados obtenidos para 100, 600 y 3000 datos etiquetados en el entrenamiento. Los datos de los modelos M1, M2 y M1+M2 de nuestra implementación se han obtenido experimentalmente. El resto de resultados fueron publicados en [6], menos los de las CNN asociativas, que hemos extraído de [5].

Como vemos en la tabla 5.1, nuestros modelos M2 y M1+M2 obtuvieron los mejores resultados. Esta diferencia es más acusada cuanto menor es el número de etiquetas disponibles en el entrenamiento, lo que le da todavía más valor a nuestra técnica, ya que con tan solo 100 datos etiquetados de un conjunto de 60000 (0.16 %), podemos reducir en tiempo y recursos el proceso de etiquetado inicial al mínimo.

Sin embargo, esto no se ha cumplido con el modelo M1 que utiliza un MLP para clasificar, que tiene el peor resultado de todos. No obstante, esto no quiere decir que las variables latentes generadas por el M1 sean incorrectas o que no codifiquen correctamente características del dato original, ya que el M1+M2 utiliza las mismas variables latentes y está entre los mejores.

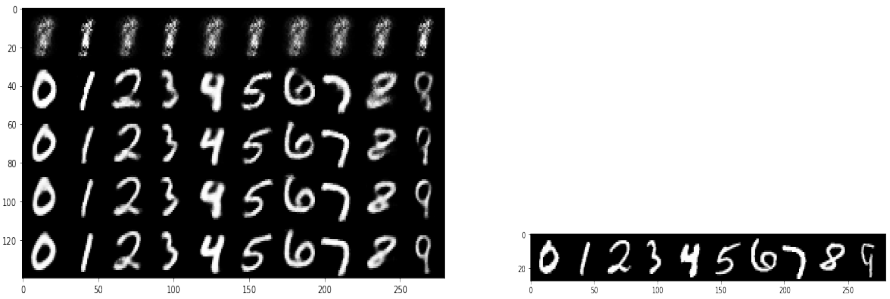
Aunque el modelo M2 también obtuvo resultados excepcionales, el M1+M2, que consistía en una implementación del M2 que tenía como entrada las variables latentes generadas por el modelo M1, fue el que arrojó los mejores resultados. Esto toma aún más importancia cuando contextualizamos el resultado teniendo en cuenta que otras aproximaciones más recientes como las CNN asociativas tienen resultados bastante peores, lo que permite afirmar que pese al paso del tiempo, los modelos propuestos se siguen manteniendo como una de las opciones más sólidas a la hora de llevar a cabo aprendizaje semi supervisado. Además, el espacio latente en el que basamos los modelos resulta ser mucho más potente a la hora de codificar información del dato original que los de otras aproximaciones como las CNN asociativas o los CAE, que también recodifican los datos, pero tienen peores resultados.

Los experimentos se han llevado a cabo utilizando una GPU Nvidia GTX 1060 en local y con la GPU Tesla K80 y la TPUv2 en Google Colab. Sorprendentemente la TPU, diseñada específicamente para realizar operaciones con `Tensor`, fue la que peor rendimiento, en cuanto a tiempo por época, obtuvo.

5.2. Análisis de los resultados del modelo M1

Para comprobar que la implementación del modelo M1 era correcta, además de contrastar que el modelo M1 + M2 obtenía buenos resultados, decidimos hacer uso del modelo generativo para generar nuevos puntos similares a los del data set MNIST. Para ello, mostramos la evolución de las imágenes generadas cada 50 épocas de entrenamiento, hasta la época 200 (figura 5.1(a)). De este modo, con cada iteración, las imágenes generadas deberían parecerse más a las originales de 5.1(b). Para hacer esto, primero tenemos que muestrear variables latentes a través del *encoder*, y después utilizarlas en el modelo generativo para crear las nuevas imágenes.

Como vemos en 5.1(a), efectivamente esto es lo que ocurre, los dígitos generados en cada época son cada vez más similares a los originales de 5.1(b). Por lo que podemos concluir que esta implementación es válida. Merece la pena comentar también que el cambio más acusado lo encontramos entre la época 0 y la 50, mientras que entre el resto de épocas, las diferencias apenas son apreciables. En general, hemos encontrado esta tendencia en los otros dos modelos, al principio los resultados mejoran de forma muy acelerada, hasta que llega un momento en el que los cambios comienzan a producirse de forma más lenta. Hemos visto este comportamiento también reflejado en la maximización del Lower Bound que vemos en la imagen 5.2



(a) Imágenes generadas por el modelo M1

(b) Imágenes originales del data set MNIST

Figura 5.1: En 5.1(a) tenemos la evolución de las imágenes generadas por el modelo M1 en orden cronológico a lo largo de 200 épocas, pintadas cada 50. La imagen de la época 0 es la de más arriba, y la más reciente (época 200), es la de más abajo. En la imagen 5.1(b) tenemos las imágenes originales.

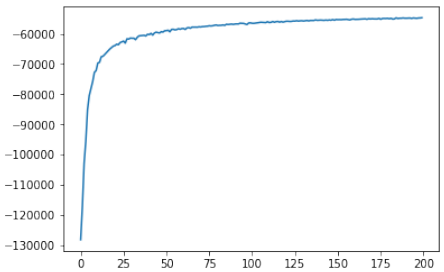


Figura 5.2: Evolución a lo largo de 200 épocas del Lower Bound en el entrenamiento del modelo M1.

5.3. Análisis de los resultados del modelo M2

En la comparativa de resultados de la tabla 5.1 vimos como el M2 sobresalía respecto a las otras técnicas del estado del arte, y solo es superado por el modelo M1+M2. Sin embargo, hay varios puntos que merecen la pena ser comentados. Por una parte, podemos decir que la implementación es correcta ya que los resultados son muy similares a los obtenidos en [6], como refleja la tabla 5.1.

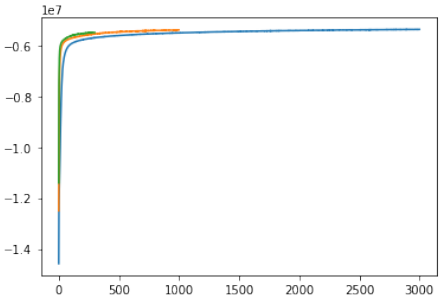
En cuanto a los datos obtenidos para 600 y 3000 etiquetas no hay mucho que añadir, los resultados son muy positivos y no encontramos ninguna característica fuera de lo normal. Sin embargo, para $N = 100$ si que podemos apreciar que la desviación típica es mucho mayor que en los otros casos. Esto no quiere decir que nuestra implementación sea incorrecta, ya que en los resultados de Kingma también nos encontramos con esta misma situación.

Respecto a la evolución del Lower Bound (figura 5.3(a)), nos hemos encontrado con que, al igual que en el modelo M1, este aumenta con mucha velocidad en las primeras épocas del entrenamiento hasta llegar a un valor casi constante, con leves cambios a lo largo de las siguientes épocas. Esta misma tendencia se observa también en el error de entrenamiento (figura 5.3(b)). En cuanto al error de test, aunque también se reduce de forma considerable durante las primeras épocas, en cierto punto sufre una pequeña subida, lo suficientemente grande como para no pasar desapercibida, y luego comienza a descender con relativa suavidad. Pudimos observar esta tendencia para todos los experimentos con diferente número de etiquetas disponibles.

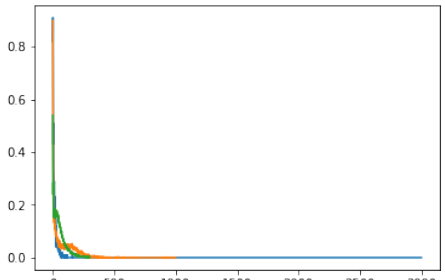
Estas gráficas nos ayudaron a estimar el número de épocas necesarias que necesita el modelo para converger. De esta forma, para obtener buenos resultados con 100 etiquetas necesitamos en torno a 3000 épocas, con 600 etiquetas, 1000, y con 3000 etiquetas nos bastó con 300 épocas.

5.4. Análisis de los resultados del modelo M1+M2

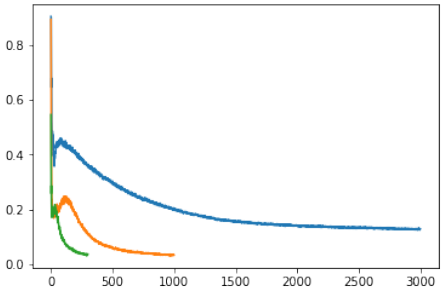
El modelo M1+M2 fue el que obtuvo los mejores errores de clasificación de nuestros modelos, y el mejor de todos para 100 etiquetas. Sin embargo, aunque los resultados de la tabla 5.1 muestran que los valores arrojados por nuestro modelo son muy similares a los de la implementación de Kingma, con 600 y 3000 etiquetas nuestros resultados son un poco peores, en concreto, alrededor de un $1{,}5\,\%$. De todos modos, el magnífico rendimiento mostrado para 100 etiquetas, nos hace ser optimistas al respecto de nuestra implementación, y pensamos que esta diferencia puede tener su origen en los parámetros elegidos referentes al peso del *weight decay* o la *learning rate*, ya que como vemos en las figuras 5.4, tanto el error de test como el Lower Bound, disminuyen su valor muy rápidamente en las primeras épocas, y después, se mantienen casi constantes. Por tanto, un regularizador del *weight decay* o la *learning rate* con distintos valores según el estadio del entrenamiento, podría resultar en mejores resultados. Por otra parte, descartamos que la diferencia se deba a las variables latentes



(a) Evolución del Lower Bound



(b) Evolución del error de entrenamiento

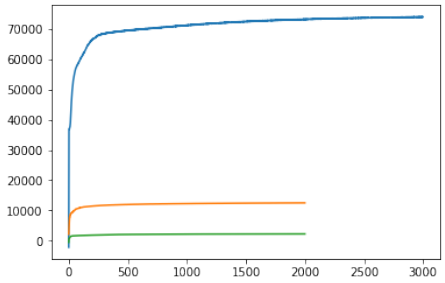


(c) Evolución del error de test

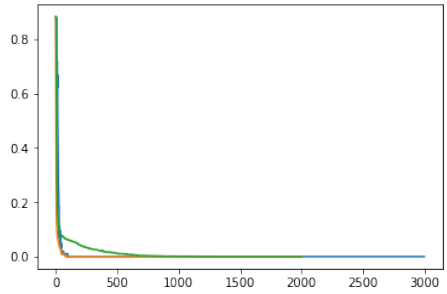
Figura 5.3: Resultados del entrenamiento del modelo M2 con 100 etiquetas durante 3000 épocas (línea azul), 600 etiquetas durante 1000 épocas (línea naranja) y 3000 etiquetas durante 300 épocas (línea verde).

generadas, ya que con 100 etiquetas, los resultados son muy buenos.

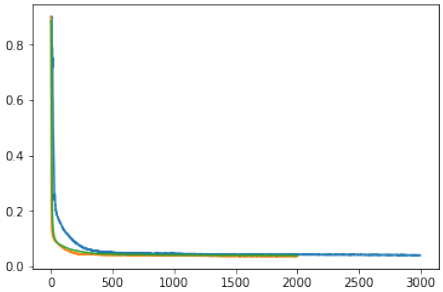
En la figura 5.4(a), vemos como las magnitudes del Lower Bound varían entre los experimentos con distinto número de etiquetas disponibles. Esto se debe a que el Lower Bound que estamos mostrando en la gráfica tiene sumada la componente del *weight decay*, que esta ponderada de forma distinta entre los tres experimentos.



(a) Evolución del Lower Bound



(b) Evolución del error de entrenamiento



(c) Evolución del error de test

Figura 5.4: Resultados del entrenamiento del modelo M1+M2 con 100 etiquetas durante 3000 épocas (línea azul), 600 etiquetas durante 2000 épocas (línea naranja) y 3000 etiquetas durante 2000 épocas (línea verde).

Cabe destacar que, aunque la complejidad de este modelo sea la suma del M1 más el M2, esto no se traduce en un tiempo de ejecución mayor, ya que como en este modelo trabajamos sobre las variables latentes, con menor dimensionalidad que el dato original, el tiempo de ejecución por época del M2 sobre $z1$ se reduce drásticamente.

6

Conclusiones

6.1. Conclusiones y trabajo futuro

Con este TFG hemos mostrado como implementar los modelos AEVB planteados por Diederik P. Kingma [6] para realizar aprendizaje semi supervisado en TensorFlow y hemos demostrado experimentalmente que los resultados que obtienen estos modelos son muy superiores a otros del estado del arte más actuales, poniendo de manifiesto que el espacio latente del que hacen uso resulta en una mejor codificación de los datos originales, y por ende, en unos mejores resultados de clasificación, y que pese al paso del tiempo, esta técnica sigue siendo una muy buena opción para realizar aprendizaje semi supervisado.

Además, esta diferencia se acentúa sobre todo cuando el número de etiquetas disponibles en el entrenamiento es muy pequeño, lo cual es una característica muy deseable, ya que permite reducir en tiempo y recursos el proceso de etiquetado inicial al mínimo, que es precisamente la motivación del aprendizaje semi supervisado.

En cuanto a trabajo futuro, tenemos pendiente rebajar el error de clasificación del modelo M1+M2 para igualarlo al de la implementación de Diederik P. Kingma, aunque la mejora final solo resulte en apenas un $1{,}5\,\%$. Por otra parte, también deberíamos investigar la causa de la alta variación entre los resultados obtenidos por el modelo M2 cuando el número de etiquetas es 100. Volviendo al modelo M1+M2, puede ser muy interesante encontrar una función objetivo que permita optimizar al mismo tiempo el Lower Bound del modelo M1 y el del M2, sería más costosa por época, pero cabría esperar que obtuviese mejores resultados.

También aportaría mucho valor optimizar el código para hacer uso de las TPU disponibles en Google Colab, ya que están diseñadas específicamente para las operaciones con `Tensor` y sin embargo en nuestra implementación con TensorFlow no vemos esta mejora, es más, el tiempo de ejecución por época es aún mayor que con GPU.

Por último, queda pendiente comprobar si los resultados obtenidos por los modelos son igual de satisfactorios en problemas de clasificación de imágenes en las que el dato original está en un espacio

dimensional mayor, por ejemplo, en el data set SVHN.

Bibliografía

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[2] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky, "Strategies for training large scale neural network language models," *2011 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2011, Proceedings*, 12 2011.

[3] S. Lohr, "The age of big data," *The New York Times*, 02 2012.

[4] O. Chapelle, B. Scholkopf, and A. Zien, "Semi-supervised learning," in *Adaptive Computation and Machine Learning*, pp. 1–3, The MIT Press, 2006.

[5] P. Haeusser, A. Mordvintsev, and D. Cremers, "Learning by association — a versatile semi-supervised training method for neural networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul 2017.

[6] D. P. Kingma, D. J. Rezende, S. Mohamed, and M. Welling, "Semi-supervised learning with deep generative models," 2014.

[7] C. Geng, Z. Yuquan, T. Jianing, and H. Tianhan, "2009 international forum on information technology and applications," in *An Algorithm of Semi-supervised Web-Page Classification Based on Fuzzy Clustering*, 2009.

[8] P. Liang, "Semi-supervised learning for natural language," 2006.

[9] X. Chen and G.-Y. Yan, "Semi-supervised learning for potential human microrna-disease associations inference,"

[10] I. Triguero, S. García, and F. Herrera, "Self-labeled techniques for semi-supervised learning: taxonomy, software and empirical study," tech. rep.

[11] E. J. C. Suárez, "Tutorial sobre máquinas de vectores soporte," 2013.

[12] T. Joachims, "Transductive inference for text classification using support vector machines," in *Icml*, vol. 99, pp. 200–209, 1999.

[13] X. Zheng, Z. Wu, H. Meng, and L. Cai, "Contrastive auto-encoder for phoneme recognition," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2529–2533, IEEE, 2014.

[14] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[15] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[16] D. Hernández–Lobato, T. D. Bui, Y. Li, J. M. Hernández–Lobato, and R. E. Turner, "Importance weighted autoencoders with uncertain neural network weights." "https://dhnzl.files.wordpress.com/2016/12/iwae_uncertain.pdf",.

[17] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2014.